Time- and Space-Efficient Spatial Data Analytics

Thèse N° 9130

Présentée le 1er février 2019

à la Faculté informatique et communications
Laboratoire de systèmes et applications de traitement de données massives
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

MIRJANA PAVLOVIĆ

Acceptée sur proposition du jury

Prof. A. Argyraki, présidente du jury
Prof. A. Ailamaki, directrice de thèse
Prof. N. Mamoulis, rapporteur
Prof. Y. Theodoridis, rapporteur
Prof. K. Aberer, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

"Life is and will ever remain
an equation incapable of solution,
but it contains certain known factors."
— Nikola Tesla

To my parents, Milka and Branko, my brother Pero,
and my aunt Boba

Acknowledgements

My PhD has been an exciting and challenging journey that would not have been possible without the help and support of my advisor, colleagues, friends, and family.

First, I would like to thank my advisor, Prof. Anastasia Ailamaki, for believing in me and for her support and guidance throughout my PhD. Natassa shaped my research skills, always setting high standards and encouraging me to do my best. Her infectious energy and enthusiasm will always be an inspiration to me.

I would also like to thank Prof. Thomas Heinis, for all his help, guidance, patience, and advice. I have worked with Thomas for the biggest part of my PhD and during these years his support has been invaluable. This thesis would not have been possible without his help.

I want to thank Prof. Yannis Theodoridis, Prof. Nikos Mamoulis, and Prof. Karl Aberer for devoting their time to serve in my thesis committee and for their insightful feedback, and Prof. Katerina Argyraki for presiding over my thesis committee.

During my PhD, I was fortunate enough to share the journey with an exceptional group of people: Adrian, Angelos, Ben, Cesar, Danica, Darius, Diane, Dimitra, Eleni, Erietta, Erika, Farhan, Foteini, Giorgos, Ioannis, Iraklis, Ivo, Lionel, Lu, Manos, Manos, Matt, Miguel, Odysseas, Panagiotis, Periklis, Pinar, Radu, Raja, Rakesh, Renata, Satya, Sharareh, Stelios, Stella, Tahir, Utku, and Viktor. They have always been there to help me with their feedback, constructive discussions or to simply chat over a cup of coffee. There are also several people in DIAS whom I owe an additional thank you to. Danica has helped me enormously in too many ways – she was there to provide advice, feedback, accommodation, and much more; she is also a great friend, with whom I shared many memorable moments. Eleni has been much more than my officemate – a friend, travel and brainstorm companion, who made my EPFL life much more fun. Giorgos has been my DIAS family in Heidelberg and unofficial officemate at EPFL, and has always given me sharp and thoughtful feedback. Manos and Matt have been my "go to" friends for advice, feedback and chat, and have always made sure to cheer me up. Stelios and Ivo made the first year of my PhD easier and less stressful by offering their help and friendship. Pinar, Renata, Iraklis, Adrian and Farhan are my academic big sisters and brothers, who have been looking out for me at all times. Finally, Erika and Dimitra have made my life much easier, by providing answers to all of my administrative questions. Thank you all for your help.

I would also like to thank my friends in Lausanne and back home who have made these years much more enjoyable. My Serbian friends, Danica, Nataša, Zlatko, Renata, Ivan, Maja, Petar,

Acknowledgements

Miloš, Marija, Darko, Aleksandar, Dražen, Peđa, Jasmina, Milena, Miloš, and Andrej have made my transition to Switzerland easy, either by helping with accommodation, or by organising a number of parties, coffees, and trips. Mickaël helped with the translation of my abstract to French. The DCL lab, Karolos, Matej, Adi, Vasilis, Giorgos, David and Tudor, made lunch time fun and interesting. Bilja and Laza welcomed me to Lausanne, and ever since then they have been looking out for me; Bilja has been my companion for trips, parties and discussions, and Laza has always given me the best advice. Bojana and Marijana made our apartment feel like home. They have always been there to encourage me and make my day better with their countless hugs. I would also like to thank my family and friends back home for being there for me, and for adjusting to my busy schedule. I owe a big thank you to Maja and Kristina, whom I have been fortunate enough to meet in the first year of my undergraduate studies; ever since then they have been my friends, my inspiration, and my daily source of positive energy.

Giorgos Chatzopoulos has made these years fun, joyful and easier. He encouraged me to do more when I doubted myself, made me laugh when I felt sad, and made me see the glass half full when I saw it half empty. Thank you for being an infinite source of love, support and positive energy, and for helping me advance in every possible aspect.

Last, but certainly not least, I would like to thank my family, my parents Milka and Branko, my brother Pero, and my aunt Boba, for supporting me in my every step and for believing in me, always. They are the ones who made me smile in the times of failures and celebrated the loudest even the smallest success. Thank you for everything you have done for me and for your unconditional love and support.

This research has been supported by the European Union Grants 604102 (FP7/2007-2013 – HBP) and 617508 (ERC-2013-CoG – ViDa), and the European Union's Horizon 2020 research and innovation programme Grants 650003 and 720270.

Lausanne, 30 December 2018 Mirjana Pavlović

Abstract

Advances in data acquisition technologies and supercomputing for large-scale simulations have led to an exponential growth in the volume of spatial data. This growth is accompanied by an increase in data complexity, such as spatial density, but also by more varied data distributions. As data evolves, so do the needs of applications. Recently, we notice a shift from predefined to ad-hoc workloads, as a result of the recent data exploration trend among data-driven applications. At the same time, given the massive volume of data, it has become imperative to use computational and storage resources efficiently, where efficiency requirements typically vary across applications.

In this thesis, we show that traditional spatial data management techniques underperform as data size and complexity increase: they waste both computational and storage resources. They are also inefficient in supporting ad-hoc workloads. To achieve time- and space-efficiency, we design spatial data management algorithms and storage layouts that leverage and adapt to data characteristics and workload access patterns. In particular, we revisit the design of spatial join algorithms, indexing techniques and point cloud data management solutions.

First, we propose data-aware spatial joins that leverage and adapt to dataset characteristics to avoid wasting computational resources and achieve time-efficiency on non-uniform data distributions. GIPSY is designed to efficiently join two datasets with contrasting densities. GIPSY uses the sparser dataset to guide the join process and therefore, by leveraging dataset characteristics, selectively retrieves and joins only the data needed. TRANSFORMERS achieves robust performance and time-efficiency on non-uniform data distributions, by adapting to dataset characteristics. It detects local variations in distributions and adapts the join strategy and data layout to local dataset characteristics at run-time.

We next introduce incremental indexing approaches that take into account workload access patterns. This way, they minimize the data-to-insight time and avoid unnecessary preprocessing costs, substantially accelerating the exploratory analysis of spatial data. Incremental indexes are built as a side-effect of query execution and only for the parts of the data queried. Space Odyssey is designed for exploratory analyses of multiple spatial datasets that reside on disk. It takes advantage of workload access patterns to incrementally index the datasets and optimize accesses to parts frequently queried together. QUASII supports spatial data exploration in main memory. QUASII reduces the data-to-insight time and curbs the cost

of incremental indexing, by gradually and partially sorting the data, while simultaneously producing a data-oriented hierarchical structure.

Finally, we propose a time- and space-efficient solution to storing and managing point cloud data in main memory column-store database management systems. Our approach leverages point cloud data properties to employ dictionary-based compression in the spatial data management domain and enhances it with indexing capabilities by using space-filling curves. The proposed scheme also represents a partitioning strategy. It is a middle ground between data- and space-oriented partitioning, accounting for the data distribution, while preserving the simplicity of grid-like structures.

Keywords: data management, database management systems, scientific data management, spatial data management, spatial data analytics, data exploration, spatial data compression, multidimensional data access methods, spatial joins, incremental indexing

Résumé

Les avancées dans les technologies d'acquisition de données et des superordinateurs pour les simulations à grande échelle ont conduit à une croissance exponentielle du volume de données spatiales. Cette croissance est accompagnée d'une augmentation de la complexité des données, comme par exemple la densité spatiale ou des distributions de données plus variées. De plus, nous remarquons une transition du prédéfini à l'ad-hoc, dû à la tendance récente de l'exploration dans les applications orientées données. De manière similaire, au vu des volumes massifs de données, il est impératif d'utiliser efficacement la puissance de calcul et l'espace disque, où les exigences diffèrent selon les applications.

Dans cette thèse, nous démontrons que les techniques de gestion de données traditionnelles sont peu efficaces avec l'accroissement de la taille et de la complexité des données : elles gaspillent de la puissance de calcul et de l' espace. Elles sont également inefficaces pour les charges de données ad-hoc. Pour atteindre un optimum en termes de temps et d'espace, nous concevons des algorithmes de gestion de données spatiales et des agencements de stockage qui exploitent et s'adaptent aux caractéristiques des données et aux schémas d'accès aux données. En particulier, nous revoyons la conception des algorithmes de jointure spatiale, les techniques d'indexage et les solutions de gestions de nuage de données.

Nous proposons d'abord des jointures spatiales s'adaptant aux caractéristiques des jeux de données. GIPSY calcule efficacement des jointures entre deux jeux de données ayant des densités différentes. Il utilise le jeu de donnée le plus épars pour guider le processus de jointure et récupère et joint uniquement les données utiles. TRANSFORMERS atteint des performances solides et optimales en termes de temps sur des distribution de données non uniformes, en s'adaptant aux caractéristiques du jeu de données. Il détecte les variations locales dans les distributions et adapte durant l'exécution la stratégie de jointure et l'agencement des données en fonction des caractéristiques locales du jeu de données.

Nous présentons ensuite plusieurs approches d'indexage incrémental sensibles à la charge de travail. Les index sont construites comme un effet secondaire de l'exécution des requêtes et uniquement pour les données étant objets de la requête. Space Odyssey est conçu pour l'analyse exploratoire des jeux de données stockés sur disque. Il exploite les schémas d'accès aux données pour incrémentalement indexer les jeux de données et optimiser l'accès au disque. QUASII supporte l'exploration de données spatiales en mémoire. Il réduit le temps

nécessaire à l'obtention des résultats et limite le coût de l'indexage incrémental, en triant progressivement et partiellement les données, tout en produisant simultanément une structure hiérarchique orientée données.

Finalement, nous proposons une solution optimale en terme de puissance de calcul et d'espace disque pour stocker et gérer des nuages de données. Notre approche exploite les propriétés des nuages de données pour utiliser une compression par dictionnaire et l'améliore en ajoutant des capacités d'indexage utilisant des courbes de remplissage. Le schéma proposé représente également une stratégie de partitionnement. C'est un compromis entre le partitionnement orienté données et celui orienté espace, qui prend en compte la distribution des données, tout en préservant la simplicité des structures de type grille.

Mots-clés : gestion de données, systèmes de gestion de bases de données, gestion de données scientifiques, gestion de données spatiales, analyse de données spatiales, exploration de données, compression de données spatiales, méthodes d'accès aux données multidimensionnelles, jointures spatiales, indexage incrémental

Contents

List of Figures

List of Tables

1 Introduction

We live in a data-driven era, with the opportunity to revolutionize science and enterprises by extracting knowledge from massive amounts of data. Data-driven science exemplary showcases the power of data: scientific simulations have become a standard practice, complementing traditional methods for understanding natural phenomena across a number of scientific disciplines [36, 41, 46, 48, 82]. At the same time, enterprises harness the insights obtained through data analyses to improve existing services and offer new ones [25, 51, 56, 135].

A significant percentage of the data collected and used has spatial properties, such as geometric extent and location. Location Based Services (LBS) provide a set of functionalities based on geographical location, used by users on a daily basis. At the same time, LBS produce massive amounts of data associated with location. For instance, Uber announced 5 billion rides in 2017 [26], while Foursquare reached 12 billion check-ins [121]. Similarly, models used in scientific simulations across a number of domains (e.g., neuroscience, astronomy, and digital pathology) are typically represented with data that is enriched with spatial extent [41, 46, 82]. With technological advancements, the amount of spatial data will only continue to increase. Internet of Things (IoT) devices are one example – they are projected to reach 20 billion by 2020 [35] and are all equipped with location intelligence.

Spatial data management systems [42, 77, 105] are specifically designed to manage spatial data, considering its spatial property (i.e., location and extent) as a first-class citizen. Recent trends in data and applications, however, challenge the design of existing solutions. Data grows exponentially in volume, evolving in complexity alongside. At the same time, a number of emerging applications introduce new requirements that challenge current solutions.

1.1 Data Evolution

As a result of the recent technological improvement, spatial data has significantly evolved in recent years, increasing in volume and complexity.

Volume. Advances in data acquisition – through more powerful supercomputers for simulations, sensors with better resolutions, etc. – have caused an exponential growth in the volume of spatial data generated and collected. For instance, in the Human Brain Project (HBP) [82], neuroscientists build spatial models of a brain, consisting of millions of three-dimensional cylinders, where several thousand cylinders together reconstruct the spatial shape of a single neuron – the final brain model is expected to reach 10^{11} neurons [127]. NASA released 500 TB of satellite earth observation data generated by remote sensing [87], while the Actueel Hoogtebestand Nederland 2 (AHN2) point cloud data set [88] contains 640 billion points acquired through airborne and terrestrial scanning. At the same time, people generate massive amounts of volunteered geographic style information (VGI). This data is acquired either through community users of services such as OpenStreetMap [94] (a free editable map of the world, created by volunteers using local knowledge), or "unconsciously" – mostly through the use of smart phones (e.g., geo-tagged tweets, Facebook check-ins, etc.).

Complexity. Alongside with volume, spatial data has also increased in complexity [126]. By improving the precision of instruments and the granularity of the models, we also increase data density (the number of objects per space unit), non-uniformity (in terms of data distribution), and irregularity in geometric shapes. This trend is particularly evident in the scientific domain. In neuroscience simulations, for instance, a neuron's morphology is modeled as a sequence of segments (cylinders) or through a fine-grain mesh representation [126, 127]. Simulations can be performed at the cellular, sub-cellular or molecular level of detail [46]. Another example is pathology image analysis, where 3D micro-anatomic objects have complex structures and often non-uniform data distribution – e.g., the tissue affected by a tumor is significantly denser, in terms of its number of cells, compared to healthy regions [5]. The increase in data complexity is not limited to the scientific domain though. In OpenStreetMap, for instance, the densest areas have nearly three orders of magnitude more objects compared to the average [5].

1.2 Driving Applications

Technological advancements have triggered not just growth in data, but also in the number of emerging applications. These applications take spatial data as input, and often also produce spatial data as output, adding to the total amount of spatial data produced. Both traditional and new spatial applications are beneficial to the scientific and business domains, where a few examples include: 1) Scientific Simulations – simulations of spatial models have become a standard practice [6, 41, 41, 82], complementing traditional methods for understanding natural phenomena across a number of scientific disciplines, 2) Digital Pathology – biomedical research is enhanced with the 3D exploration of micro-anatomic objects [71, 72, 73], 3) Urban Planning and Smart Cities – aerial scanning, combined with remote sensing technologies, enable planning and monitoring of the urban development process [1, 23, 31], and 4) Geographic Information Systems (GIS) – systems designed to store and process geographic data naturally use massive amounts of spatial data and play an important role in natural resource and disaster management [32, 119], telecommunication and network services [115], and more.

As data volume and complexity increase, such applications become the main means that enable users to extract useful information from it. As such, these applications have potential to advance science and enterprises by enabling new discoveries, inspiring new products and improving existing services. The performance of such applications, and thus their usefulness, is determined by their ability to process massive amounts of spatial data *efficiently*. By efficiently, we refer to processing of data in a *time-* and *space-efficient* manner.

Time-efficiency. Given the massive volume of data, it has become imperative to use computational resources efficiently, i.e., to maximize performance and provide timely response. Applications have always wanted performance, however, as data size and complexity increase, this has become a critical requirement. For instance, Uber has to process user requests and provide services, such as dynamic pricing, in near real-time [27]. Urban planning applications rely on visual analytics systems which are expected to provide interactive response times [23, 139]. Finally, to advance scientific discovery, scientists rely on tools that enable fast processing of massive amounts of spatial data [46, 47, 52, 123, 126].

A number of applications focus on identifying and extracting useful insights from data, having the "usefulness criteria" defined at runtime. Their analysis is data-driven: users do not know a priory what they are looking for, and determine future steps based on the results of previous analyses. This new class of applications is described with a newly-coined term – data exploration [7, 15, 52, 55, 126]. Performing data exploration requires extracting knowledge from data in a timely manner [7, 55, 126]. More specifically, users need to analyze data the moment it is available. Otherwise, data could lose some or all of its value – which, by the time they discover it, might have already cost them significant time and processing resources.

Space-efficiency. The increasing amount of stored and managed data demands for space-efficient data organization. Being space-efficient implies reduction in terms of resources necessary to store and transfer data. The primary motivation for the space-efficiency requirement are the massive amounts of data that both scientific and enterprise applications have to store and manage today (Section 1.1). To ensure resilience to errors, this data gets additionally replicated, increasing the total amount of data necessary to be maintained. Storage resources are cheap, however, their capacity cannot keep up with the exponential growth of data. Therefore, to handle this data deluge, we have to reduce space requirements and thus, the overall cost of systems. Alongside with storage reductions, space-efficient data organization has also the potential to improve performance [3, 4, 109, 141].

1.3 Data Management Challenges

To meet the aforementioned requirements and fulfill their purpose, applications depend on efficient spatial analytics data management. Spatial data management has been an important research direction for more than four decades [33, 42, 77, 105, 112]. Traditionally, processing algorithms and supporting data structures have been designed specifically for spatial data to maximize performance benefits, taking into consideration its spatial property. Recent

technological advancements, however, make this insufficient and challenge the design of existing solutions and therefore, their ability to meet time- and space-efficiency requirements.

Keeping Up with the Data Evolution. Increases in data volume and complexity challenge the time-efficiency and scalability of existing solutions. By becoming bigger, data gets also denser and more non-uniform with respect to data distribution, as discussed in Section 1.1. Due to the changes in data properties, existing problems, which are common for spatial algorithms and data structures, get exacerbated, and new ones appear.

For instance, the overlap between nodes in data-oriented structures (i.e., the property of data-oriented partitioning, but also its challenge [33, 113, 130]) increases alongside with the increase in density – causing significant performance penalties [127]. Similarly, given the non-uniformity property, achieving robust performance with respect to the distribution of data is important in order to stabilize and optimize performance. The design of traditional data management solutions, however, is mostly agnostic to evolving data properties, i.e., it does not accommodate for these properties, incurring significant performance penalties [90, 127, 128]. Spatial join, a core operator in spatial analytics [58, 76, 78], is particularly affected by these changes, given its default high cost compared to the other types of spatial queries [76].

Supporting Ad-hoc Analysis. To conform with the data exploration requirements, we have to analyze data as soon as it is available, and to extract useful information quickly, in an ad-hoc manner. Traditional systems do not fulfill these requirements, as they require expensive preprocessing steps and a priori workload knowledge to meet the expected performance. Consequently, they significantly increase the data-to-insight time. Additionally, they can potentially waste both computational and storage resources, if the preprocessing cost is not amortized with the subsequent data analyses.

Addressing the aforementioned challenges, incremental and adaptive data processing forms the core of efficient data exploration [52, 55]. To efficiently identify and extract useful information from massive amounts of data in an ad-hoc manner, algorithms and index structures have to be incremental, and adapt to the changes in data and workload seamlessly. While incremental and adaptive data processing have been extensively studied in the relational domain [7, 45, 52, 53, 54, 55], they have been overlooked in the spatial domain, resulting in a lack of support for data exploration tasks. Traditional spatial data management systems require indexes to be built before analytical queries can be executed efficiently. Therefore, support for incremental indexing is key to efficient spatial analytics in an ad-hoc environment.

Supporting Emerging Features. The evolving data properties and application requirements combined demand support for new features. Traditional solutions are not designed according to the emerging requirements and data properties, and therefore, cannot typically support new functionality efficiently – such that the data is used to its full potential and the demanding applications requirements are met. A driving force for new features is usually scientific discovery, where technological advancements and new use cases (novel methodologies) challenge the applicability of existing solutions [126, 128]. Business applications are equally relevant,

as they leverage new data sources to improve existing services or offer new products, where point cloud data management is a recent example [8, 37, 124, 134].

With the advances in laser and image processing technology, data properties and users expectations in terms of point cloud management have evolved [124, 134], challenging the efficiency of traditional file-based solutions. Data has increased significantly in size and precision. At the same time, applications want to use this data to its full potential: combining it with the other types of data, while using declarative and ad-hoc queries to explore it [8, 124, 134]. As a consequence, point cloud processing moved toward database management systems that are expected to provide both time- and space-efficiency, given the size of data.

1.4 Thesis Statement and Contributions

Spatial data analytics represent a powerful means to extract knowledge from data, however, due to recent technological advancements, there is a significant discrepancy between expectations (determined by application requirements) and the actual ability of data management systems. The goal of this dissertation is to help advance spatial data management systems to reach their full and expected potential, by revisiting the design of traditional spatial data management systems and advocating for changes that will bridge the gap between the requirements and actual system performance.

Thesis Statement

Traditional spatial data management techniques underperform as the data size and complexity increase: they waste both computational and storage resources. Time- and space-efficiency of analytics is improved if spatial data management algorithms leverage and adapt to data characteristics and workload access patterns.

We revisit the design of spatial join algorithms, indexing techniques, and point cloud data management solutions based on the following key insights:

- **Adapt to Data.** Being agnostic to data characteristics and employing static strategies leads to sub-optimal, non-robust performance when joining datasets with non-uniform distributions. The key to optimize performance is to be data-aware, that is to adapt the join strategy and supporting data structures to the underlying data distributions, in order to maximize performance and avoid wasting computational resources.
- **Adapt to Workload.** Workload-driven incremental index building significantly reduces the data-to-insight time. Indexes are built as a side-effect of query execution, and only for the parts of data queried, reducing computational and storage requirements. To ensure efficiency, incremental indexing should meet the following requirements: minimal data-to-insight time, efficient query performance, and low-cost incremental strategy.

- **Leverage Data Characteristics.** Preserving spatial proximity through data organization has the potential to improve time- and space-efficiency, as 1) data access patterns are frequently aligned with spatial proximity, i.e., objects close in space are frequently processed together, and 2) compression techniques can exploit spatial proximity. To maximize performance, it is equally important to leverage secondary data characteristics, other than spatial. These characteristics are typically introduced as a result of technological advancements and introduce new patterns in data that should be exploited.

Based on these insights, this dissertation makes the following technical contributions:

- We present the design and implementation of two disk-based spatial join approaches that leverage and adapt to dataset characteristics, achieving time-efficiency on non-uniform data distributions.
  - GIPSY is a spatial join approach designed to efficiently join two datasets with contrasting densities. GIPSY uses the sparser dataset to guide the join process and therefore, by leveraging dataset characteristics, it selectively retrieves and joins only the data needed. GIPSY relies on data-oriented partitioning to produce fine-grain partitions, while avoiding overlap-related problems, by employing a crawling strategy.
  - TRANSFORMERS achieves robust spatial joins on non-uniform data distributions, by adapting to dataset characteristics. We first show that the performance of the state-of-the-art spatial join approaches deteriorates when faced with variations in the distributions of data. To achieve robust performance and time-efficiency, TRANSFORMERS detects local variations in distributions and adapts the join strategy and data layout to local datasets characteristics at run-time.
- To achieve timely response and thus provide tools for efficient spatial data exploration, we design and develop two incremental indexing techniques. Both approaches take advantage of workload access patterns to significantly reduce data-to-insight time and achieve query performance comparable to traditional indexing approaches.
  - Space Odyssey is designed for exploratory analyses of multiple spatial datasets that reside on disk. Without any prior information, Space Odyssey incrementally indexes the datasets and optimizes accesses to parts of the datasets frequently queried together, adapting their data layout on disk to the workload access patterns.
  - QUASII is a query-aware spatial incremental index, designed for exploratory analyses of spatial data in main memory. It reduces data-to-insight time and curbs the cost of incremental indexing by gradually and partially sorting the data, while simultaneously producing a data-oriented hierarchical structure. In addition to QUASII, we also demonstrate the challenges of adapting and using existing incremental approaches (designed to incrementally index relational data) in the spatial domain.
- To accommodate for the recent requirements and adjust to evolving data properties, we present the design and implementation of a time- and space-efficient solution to storing and managing point cloud data in main memory column-store DBMS. We leverage point cloud

data properties, i.e., the frequent repetition of values for the *x*, *y*, and *z* coordinates across point cloud entries, to employ dictionary-based compression in the spatial domain and thus achieve space-efficiency. To optimize for time-efficiency, we enhance dictionary-based compression with spatial indexing capabilities. More precisely, we integrate a space-filling curve order into the dictionary-based model, without requiring additional storage resources.

1.5 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 presents related work and background information on the topics of this thesis.

Part I presents data-aware spatial joins, i.e., we introduce two spatial join approaches that leverage and adapt to dataset characteristics to achieve time-efficiency. Chapter 3 introduces GIPSY [99], a spatial join approach designed to efficiently join two datasets with contrasting densities. GIPSY uses the sparser dataset to navigate the join process and therefore, by leveraging dataset characteristics, it selectively retrieves and joins only the data needed. Chapter 4 presents TRANSFORMERS [100], a spatial join approach that achieves robust performance and time-efficiency on non-uniform data distributions by adapting to dataset characteristics. TRANSFORMERS detects local variations in distributions and adapts the join strategy and data layout to local datasets characteristics at run-time.

Incremental indexing enables timely response and saves both computational and storage resources by building an index as side-effect of query execution and indexing only the parts of the data queried. While incremental indexing has been extensively studied in relational domain, it has not been addressed in spatial data management. Part II introduces two query-aware incremental indexing approaches designed for exploratory analyses of spatial data. Chapter 5 presents Space Odyssey [101], designed for exploratory analyses of multiple spatial datasets that reside on disk. Space Odyssey takes advantage of workload access patterns to incrementally index the datasets and optimize the access to datasets frequently queried together. Chapter 6 introduces QUASII [103], a query-aware spatial incremental index, designed for exploratory analyses of spatial data in main memory. To reduce data-to-insight time and curb the cost of incremental indexing, QUASII relies on a partial sorting strategy and data-oriented hierarchical structure.

Part III presents a time- and space-efficient solution to storing and managing point cloud data in main memory column-store DBMS, motivated with the recent requirements and evolving data properties in point cloud data management. Our approach [102] (Chapter 7) leverages point cloud data properties to employ dictionary-based compression in spatial domain and enhances it with indexing capabilities to achieve both time- and space-efficiency.

In Chapter 8 we summarize the thesis and discuss future work directions.

2 Background

In this chapter, we present a brief overview of topics that are related to this thesis, including a survey of related work. We begin by describing the key concepts in spatial data management. We then discuss traditional spatial access methods, designed for efficient querying of spatial data. We continue by giving an overview of the state-of-the-art in terms of spatial join techniques. Finally, we provide background on point cloud data management.

2.1 Fundamentals of Spatial Analytics

Spatial data management has been an important research direction for more than four decades [33, 42, 77, 105, 112]. Processing algorithms and supporting data structures have been designed specifically for spatial data, considering its spatial property as a first-class citizen. Two major properties that distinguish spatial from one-dimensional data and make its management challenging are: 1) a complex structure, and 2) its lack of total order. In the following we discuss in more detail these challenges and their implications [33, 77, 105]. We also address the general design behind spatial algorithms and supporting data structures and outline the common type of spatial queries.

2.1.1 Spatial Data Representation

The term spatial object has a broad sense – it can represent a point, line, polygon, spherical object, cube and more. Overall, we can categorize spatial objects into two categories: points and extended objects. A point is represented with its location in a two- or higher-dimensional space. Extended objects, such as lines, regions/polygons, or volumetric data (in 3D space), are also identified with a geometric extent that represents the geometric shape of the object. Therefore, a spatial object can represent a single point or a polygon defined by hundreds or thousands of points. Consequently, it is challenging to store, index, and process spatial data efficiently given its complex structure and varying size.

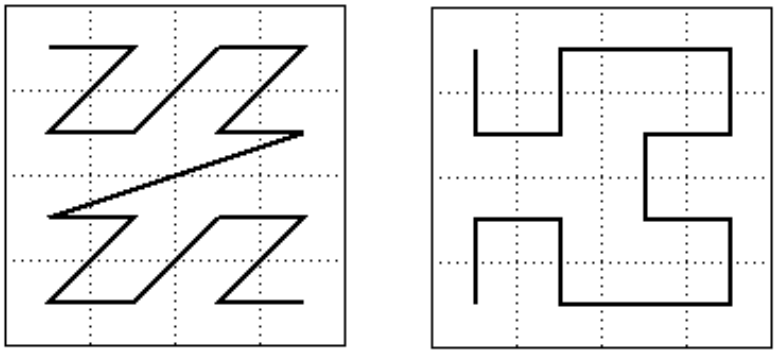


Figure 2.1 – Space-filling curves: Z-order and Hilbert curve.

To address the aforementioned challenge, spatial data is typically processed in two steps: filtering, followed by refinement [19, 33, 58]. In the filtering phase, algorithms work with approximations of the spatial objects. These approximations have significantly simpler structures than the actual objects, and therefore they primarily minimize the computational cost. The goal of the filtering phase is to maximize the amount of data that can be processed using just the approximation geometry. This phase results in a candidate set – a list of objects that satisfy the spatial predicate given the approximation geometry. Finally, the refinement phase examines the actual objects from the candidate set to remove any false positives detected due to the use of approximation.

A typical object approximation is a minimum bounding rectangle (MBR) or a minimum bounding box (MBB), depending whether they represent 2D or 3D data. A MBB of an object is the smallest box that encloses the complete geometry of an object and it has iso-oriented sides. It is the most frequently used approximation because of its simple structure that minimizes computational and storage requirements. One of the issues associated with the approximation structures is the problem of dead space - a portion of space that is marked with the object's approximation, but not occupied by the object. Several techniques [16, 17] have been designed to minimize dead space in an object's approximation and consequently improve the filtering stage. However, the overall benefit of these techniques might not be significant if they result in more complex structure that increases the storage requirements, the cost of pre-processing and the intersection tests.

2.1.2 Linear Orderings

One of the major challenges encountered when managing spatial data is the lack of a total order among spatial objects that preserves spatial proximity. More precisely, there is no total order that guarantees that any pair of objects which are close in the higher-dimensional space will also be close in the total order. Consequently, algorithms that use data sorting at their core cannot be directly employed for spatial data processing.

One way to introduce a notion of total order among spatial objects is by means of space-filling curves, such as the Z-order [96], the Hilbert curve [59], and the Gray-code curve [28]. A space-filling curve imposes a total, 1D order by visiting all the partitions in a D-dimensional grid exactly once, as illustrated in the Figure 2.1. The order in which the partitions are visited defines their 1D codes and the order in 1D space. The granularity of the grid defines whether each object or group of objects will be assigned with a unique 1D code.

Given that it introduces the notion of total order, a space-filling curve also maps data to a one-dimensional domain, as it assigns 1D codes to spatial objects. Given such a mapping of spatial data, existing 1D access methods and algorithms can be used to manage spatial data. For instance, a B-Tree [11] can be used by transforming both data and queries in a 1D domain.

It is important to note that the dimension reduction can introduce performance penalties. First, given that there is no natural total order among multi-dimensional objects, space-filling curves preserve spatial proximity up to different extents – depending on the employed ordering schema. Therefore, when mapping spatial data, it is crucial to consider space-filling curves that are effective in preserving spatial proximity [29, 59, 86], such as the Hilbert curve [59] and the Z-order [96]. Second, in order to achieve efficiency, the corresponding mapping schemes have to be adjusted not to introduce overheads with the transformation to 1D space. One example is range query mapping – if a simple mapping is considered, the transformed 1D range can be significantly larger than the original multi-dimensional range. Techniques that partition the curve into multiple sub-intervals, each of which is fully contained in the original range [132], address this issue.

2.1.3 Native Spatial Data Organization

Space-filling curves introduce the notion of total order, and consequently enable the use of 1D access methods and algorithms in spatial data management. While the dimension reduction provides simplicity, it also reduces the level of information and does not offer the same flexibility as the native, multi-dimensional domain. Therefore, a number of algorithms and data structures have been designed specifically for spatial data. One of their main design goals is to organize data such that spatial proximity is preserved. Preserving spatial proximity is important, as it has the potential to improve time- and space-efficiency, given that 1) data access patterns are frequently aligned with spatial proximity (i.e., objects close in space are frequently processed together), and 2) compression techniques can exploit spatial proximity.

Considering the type of data partitioning (i.e., data organization) they employ, spatial algorithms and supporting data structures can be divided in two categories – approaches based on space- and data-oriented partitioning. The partitioning technique determines the benefits, but also the drawbacks for the corresponding family of approaches.

Space-oriented Partitioning. Space-oriented partitioning is done by partitioning the space containing the data, regardless of the spatial distribution of the objects [57, 65, 98, 111]. The

produced partitions are disjoint, i.e., they do not overlap. Consequently, space-oriented partitioning entails a low-cost pre-processing step when it comes to identifying the partitions that contain the data of interests, as it typically employs a D-dimensional hash function.

While space-oriented partitioning provides simplicity, it also limits the ability to adjust to the data distribution. More precisely, it does not provide explicit control over the number of elements assigned per partition and therefore, it can exhibit difficulties when handling skew. However, the major drawback of space-oriented partitioning is related to the handling of volumetric objects[1]. Given that the produced partitions do not overlap, a volumetric object can intersect with several partitions. To address this ambiguity, partitioning approaches use a multiple assignment or a multiple matching strategy.

The multiple assignment strategy assigns a copy of the object (or a reference) to each partition the object intersects with [89, 98]. Doing so has the advantage that, given a point query, we can uniquely identify a partition that the query overlaps with or follow a single path while traversing a hierarchical data structure. Replicating objects, however, has several disadvantages: 1) results may be detected twice and deduplication is required or on-line duplicate removal [22], 2) the same object can be considered multiple times for the corresponding predicate check, 3) more data has to be stored and transferred, and 4) the grid configuration becomes more challenging as increasing the number of partitions also increases the replication rate.

The multiple matching strategy avoids replication of objects and assigns each object only to one partition it intersects with [9, 65]. The corresponding data structure is equivalent to a hierarchy of space-oriented structures of increasing granularity. To partition the data, each object is assigned to the lowest level in the hierarchy, where it only overlaps with one partition. While this approach to partitioning avoids replication, it requires the inspection of multiple grids (that share a border).

Considering that the major drawback of space-oriented partitioning is related to the handling of volumetric objects, this type of data partitioning is mostly used to organize points, as they do not encounter similar issues. More precisely, while a volumetric object spans multiple disjoint partitions, a point is assigned to exactly one partition.

Data-oriented Partitioning. The data-oriented strategy partitions the data taking into consideration its spatial distribution [12, 18, 43, 74]. The produced partitions consequently adjust to the data distribution, and control space utilization by limiting the number of objects assigned per partition. Adjusting to the data distribution is important, as it improves 1) handling skew in the data, and 2) data filtering capabilities – for instance, in the spatial join process we can use the partitions of one data set to build an index on the other dataset and/or to prune irrelevant elements. The control of space utilization is particularly useful for disk-based approaches, as it allows packing a fixed number of elements per partition.

[1] We work with 3D data and therefore, refer to objects with spatial extent as volumetric objects.

The main design characteristics of data-oriented partitioning are that 1) an object is assigned to just one partition, and 2) partitions can overlap. Assigning an object to one partition eliminates problems related to multiple assignment and multiple matching strategies. Consequently, data-oriented partitioning is preferable when it comes to managing volumetric objects.

However, assigning an object to just one partition in a data-oriented manner has also its drawbacks, as it results in overlaps among partitions that can degrade performance. More precisely, given a point query, we cannot uniquely identify the partition that the query overlaps with or follow a single path while traversing a hierarchical data structure. To address the problem of overlap, several approaches partition the data by producing non-overlapping regions. However, to be able to achieve so, they allow object replication and consequently encounter similar problems to space-oriented partitioning approaches.

2.1.4 Spatial Queries

A number of approaches have been designed to address efficient spatial query execution. In the following we outline the common query types used for spatial data analyses.

- **Range Query.** A range query is the most common type of spatial query. Given a dataset A, a spatial region r (or a reference object o), and a predicate θ, it retrieves all the objects from the dataset A that satisfy the spatial predicate θ with respect to the region r (or a reference object o) [33, 77]. The spatial predicate can be defined with any spatial relationship between objects, such as intersection, enclosure, or containment. The region can be explicitly defined (typically as an iso-oriented region), or inferred based on the reference object. For instance, by using a range query we can inspect the properties of a spatial model given a specific region in the space, or find all hotels within 500 meters from a user's current location.
- **Nearest Neighbor Query.** Given a dataset A and an object o, a nearest neighbor query returns all the objects from dataset A that have the minimum distance from the object o. Formally defined, $NNQ(o) = \{a \in A : \forall a' \in A, dist(a,o) \leq dist(a',o)\}$ [33, 77]. One variant of the nearest neighbor query is the k-nearest neighbor query that returns the k nearest neighbors. For instance, finding the 5 restaurants closest to a user's location.
- **Spatial Join Query.** Given two datasets of spatial objects A and B, and a predicate θ, a spatial join query returns the pairs of objects that satisfy the spatial predicate. Formally, $A \bowtie_\theta B = \{(a,b) : a \in A, b \in B, a\,\theta\,b\}$. The efficient execution of spatial join queries is important in many different applications. In scientific applications, for example, spatial joins are used to determine the location of synapses in brain models, in medical imaging to determine the proximity of cells, and in geographical information systems spatial joins detect collisions between geographical features like houses, roads, etc.

2.2 Spatial Indexing

Research in indexing spatial data has produced numerous approaches for the fast and scalable querying of spatial datasets [33]. We first discuss the traditional approach to indexing, where an index is built as part of pre-processing. We then introduce the concept of incremental indexing and related approaches.

2.2.1 Traditional Indexing

We briefly review traditional spatial indexing approaches that we group into two classes according to the division introduced in Section 2.1.3. Each class inherits the benefit and drawbacks characteristic of the corresponding family of approaches. We do not distinguish between methods for points and volumetric objects. The approaches that are designed for points can be adopted to handle volumetric objects using the query extension technique [122]. The basic concept is to represent a volumetric object by its center, and use a query enlarged by the size of the biggest object [122]. However, as discussed in Section 2.1.3, approaches based on space-oriented partitioning are typically used to manage point data, while data-oriented partitioning is a preferable choice for volumetric objects.

Space-oriented Indexing. A typical representative of approaches based on space-oriented partitioning is the uniform grid that partitions the space uniformly into partitions of equal size [114, 116], employing the multiple assignment strategy. Similarly, the Quadtree [111] and its variant for 3D space, the Octree [57], recursively divide space into four/eight partitions of equal size to build a hierarchy of partitions.

Further approaches, for example the grid file [89], use a non-uniform grid to better accommodate skew in the data (and to optimize for disk accesses). The downside is a more complex query execution, due to cells of different sizes and locations. The two level grid file [49] addresses the issue by introducing an additional level with a coarser grid. Still, the overhead of testing a query against multiple cells can be substantial. Further work improves on the space utilization by adding a second grid [50]. BLOCK [92] is a recent in-memory index designed to reduce the number of intersection tests by adjusting the granularity of the grid to the spatial properties of the query. To achieve this, it creates a set of grids of different granularity, and splits each query across the available grids.

A space-filling curve [28, 59, 96] uses space-oriented partitioning at its core. More precisely, the corresponding curve is constructed by ordering the cells of a uniform grid. Given that the space-filling curve transforms spatial data from a multi- to a one-dimensional domain, existing 1D access methods, such as the B-Tree [11], can be used for querying.

Data-oriented Indexing. The R-Tree [43], arguably the most important data-oriented spatial index, is a multi-dimensional generalization of the B-Tree, which recursively encloses objects in MBRs. The basic R-Tree definition faces the problems of overlap and dead space, both

detrimental to query execution performance [43, 127]. Multiple approaches have been devised to address the issue. The R*-Tree [12], for example, uses an improved version of the node split algorithm and reinsertion of objects, while the R+-Tree [113] tries to avoid overlap through the duplication of MBRs. A priori knowledge of the entire dataset may help to reduce the above problems of the R-Tree by packing spatially close objects on the same disk page. The Hilbert R-Tree [61] achieves this using the Hilbert curve, Sort-Tile-Recursive (STR) [70] recursively sorts objects in all dimensions to do so, while Top-down Greedy Split (TGS) [34] recursively splits the data set into partitions, minimizing the area on each level.

Adaptive index structures [125] are designed to optimize disk-access for data-oriented indexes (including the R-Tree index), based on the query workload. The core idea is to rearrange the index nodes in response to queries, so that they can be accessed sequentially on disk. While most of the R-Tree-based indexes are designed for disks, the CR-Tree [64] optimizes R-Trees for main memory access. The CR-Tree is a cache-conscious version of R-Tree. To reduce the number of cache misses in R-Tree, CR-Tree proposes a MBR compression scheme that reduces the index size and enables the alignment of the nodes to the cache lines.

The KD-Tree [13] is a binary search tree that recursively divides space, using an iso-oriented hyperplane to divide the space into two parts at every step. The adaptive KD-tree [14] extends the initial KD-Tree design. It takes into account data distribution by splitting the space into two partitions, such that each partition contains approximately the same number of points. While the KD-Tree is designed for points, the SKD-tree [93] or spatial KD-Tree is designed for volumetric objects.

2.2.2 Incremental Indexing

Traditional systems require indexes to be built before analytic queries can be executed efficiently. More precisely, they require sufficient idle time for an index to be build before any kind of analyses can be performed. Modern, data-driven applications, however, do not conform to these requirements. Their goal is to analyze data as soon as it is available, i.e., to minimize data-to-insight time.

The concept of incremental indexing addresses this challenge: to reduce data-to-insight time, an index is built as a side-effect of query execution and only for the parts of the data queried. To the best of our knowledge, no incremental strategy has been proposed for spatial indexing. Nevertheless, there has been considerable interest in incremental data processing within relational databases. We thus briefly describe approaches used by relational database systems.

Incremental indexing techniques have been extensively studied in database cracking [45, 53, 54] and adaptive merging [38, 39]. Both categories of techniques distribute the cost of sorting across the queries. The former partially sorts keys in an in-memory relation, essentially performing quicksort. The latter, adaptive merging, takes the idea further and devises an

incremental, external sort to make use of external memory as well. Similarly, incremental approaches can be used to index time-series [140].

Driven by the same goal, novel systems have been proposed that bypass the data pre-processing step and execute queries on raw data files. Instead, auxiliary data structures are built incrementally so that the most popular data subsets are serviced at the speeds of fully loaded/indexed data. For example, NoDB [7], RAW [62], and ViDa [63] incrementally build positional maps to track the position of frequently accessed data fields. This enables these systems to "jump" to previously queried data regions and potentially reduce the costs of tokenizing and parsing raw data sources. Following the same paradigm, Slalom [91], an in-situ query engine, provides on-the-fly partitioning and indexing schemes based on the information collected by lightweight monitoring.

2.3 Spatial Joins

Spatial join is one of the fundamental operators in spatial analytics [58, 76, 80, 131]. Given two datasets of spatial objects, a pairwise spatial join finds the pairs of objects that satisfy the spatial predicate. The predicate can be defined with any spatial relationship between objects, such as intersection, enclosure, or distance – where intersection/overlap is the most commonly used. Given its importance, a number of algorithms have been developed to perform spatial joins, where the majority is designed for disk. Following the division introduced in Section 2.1.3, we categorize the approaches according to the use of space- or data-partitioning strategies.

Space-oriented Partitioning. We organize join methods based on space-oriented partitioning into two groups according to the use of a multiple assignment or multiple matching strategy.

PBSM [98], the partition based spatial-merge join, partitions the space uniformly into cells of equal size, employing the multiple assignment strategy. In the first phase, each element of both datasets is assigned/replicated to all cells it overlaps with. In the second phase, PBSM iterates over all cells $c \in C$ and tests all elements of dataset A in c against all elements of dataset B in c to find pairwise intersections.

The size separation spatial join (S3) [65] uses a hierarchy of equi-width grids of increasing granularity. Each element of both datasets is assigned to the lowest level in the hierarchy where it only overlaps with one cell. To perform the join, S3 iterates over each cell c in the hierarchy and joins it with all cells that cover c on a higher level, following the concepts of the multiple matching strategy.

The scalable sweeping-based spatial join [9], a representative of the multiple matching strategy, partitions the space into n strips of equal width in one dimension and assigns each element e of both datasets to the strip where e is fully contained. In each of the n strips it uses a plane-sweep approach to determine all intersections between elements of datasets A and B.

Elements intersecting with several strips (e.g., from strip i to strip k) are assigned to set S_{ik}. When swiping strip j all sets S_{jk} with $j < n < k$ are also joined.

Data-oriented Partitioning. We organize joins based on data-oriented partitioning into three categories, depending on whether they require an index on none, one, or both datasets.

The synchronized R-Tree traversal [18] synchronously traverses the R-Trees [43] R_A and R_B built based on datasets A and B. Starting at the root nodes of R_A and R_B, the approach traverses the trees top down and, if two nodes $n_A \in R_A$ and $n_B \in R_B$ on the same level intersect, recursively tests their children. On the bottom level, the spatial elements are tested for intersection. Optimization techniques, such as search space restriction, or sorting based on the plane sweep strategy [18], can be used to reduce the cost of intersection tests.

While the synchronized R-Tree traversal requires both datasets to be indexed, the indexed nested loop join [24] only requires an index I_A on dataset A. It iterates over dataset B and queries I_A for each element $b \in B$ with b as the query. The query results are all intersections of b in A. Given the considerable cost of a query, this approach is only efficient in case $A >> B$.

Several approaches take advantage of existing indexes by using seed-style strategies. The seeded tree approach [74] assumes the existence of an R-Tree index I_A built based on dataset A. It uses I_A as a template to build a second R-Tree index I_B based on dataset B. Both indexes are joined with the synchronous R-Tree traversal [18] approach. As I_B is built based on I_A, the bounding boxes are aligned leading to less overlap and the synchronous join therefore has to compare fewer bounding boxes. The slot index spatial join [79] uses the existing R-Tree index I_A as partitioned data, i.e., to define a set of partitions or slots. The slots are used to partition the non-indexed dataset B, applying simultaneously data filtering – the objects that do not intersect with any slot are filtered. In the final phase, the corresponding partitions are joined.

The spatial hash join [75] similarly applies the seed-style strategy, however, it does not assume the existence of indexes. It uses sampling to hash dataset A into a predefined number of buckets. The produced buckets are then used as a seed to partition dataset B. While the seed strategy enables data filtering, it also produces partitions that might not be reused given that parts of dataset B are filtered out.

2.4 Point Cloud Data Management

Point cloud data is a useful source of information for natural resource management, urban planning, architecture and more. It represents a set of 3D points used to model an object or area, exploiting a number of properties such as x, y, and z coordinates, angle of scan, and color. We first give a general overview of existing point cloud management systems, following up with compression and indexing capabilities of current systems.

General. File-based solutions (e.g., LAStools [108]) represent a traditional approach to point cloud data management: points are stored in files in a predefined format and processed by

application-specific algorithms. While file-based solutions have been widely used, as point cloud data increases in size and popularity, it becomes more challenging for them to meet recent data management requirements. First, file-based solutions have limited functionality in terms of declarative power and multi-user support [134]. Second, they face scalability problems with respect to the increasing number of files to process and their size [8, 134]. A recent benchmark [134] proposes a hybrid solution to address the scalability problem, employing a DBMS to manage the meta-data of a file-based solution.

Research in this area has recently shifted towards DBMS as many of the data management challenges encountered with the increasing point cloud data size, have already been addressed in DBMS solutions. Current DBMSs support point cloud data management in the form of extensions and specific data types, distinguishing between the blocks and flat table models. The blocks model groups spatially collocated points into blocks, preserving spatial proximity. Although the blocks organization offers basic compression capabilities, it also requires blocks to be unpacked when executing queries. This introduces overhead when executing high selectivity queries [134]. On the other hand, the flat model uses the straightforward approach of storing one point per row, which provides simplicity, however, it requires significant storage resources [134]. A recent benchmark [134] evaluates the performance of these models, experimenting with various systems, including both file-based solutions and DBMSs. More precisely, the blocks storage model was tested through Oracle and PostgreSQL (their point cloud extensions [95, 106]), while the flat model was used in MonetDB, but also in Oracle and PostgreSQL. The workload (mostly selection queries) was defined based on users requirements [124].

Recent work [8, 67, 134] illustrates the potential of column-store DBMSs to meet point cloud data management requirements. The MonetDB demo [8] showcases the declarative power of DBMS when managing point cloud data, enriched with semantics from different data sources. On the other hand, the approaches proposed in [67] focus on improving the existing algorithms for spatial selections and joins on modern hardware in the context of point cloud data management.

Compression. The blocks model offers basic compression capabilities as the points within a block have a common base. For instance, PostgreSQL and LAS represent the point cloud entries within each block as integers with a scale and offset value. An alternative option in PostgreSQL is dimensional compression where each dimension is separately compressed using algorithms such as run-length encoding. In [84], the authors propose a compression scheme for the flat storage model in MonetDB. Morton-replacedXY [84] compresses data by representing a point with a z coordinate and Morton code that replaces the x and y coordinates.

Indexing. Both file-based solutions and DBMSs (based on the blocks model) by default organize data to preserve spatial proximity information and thus optimize query execution. This has been mostly done by using space-filling curves, such as the Hilbert curve [59] and the Z-order [96]. To further optimize performance, they use index structures such as R-Tree [43], octree [57], quadtree [111] etc. The flat model does not preserve spatial data properties by

default, as it stores the x, y, and z coordinates independently. Therefore, one option is to treat data as non-spatial and thus use indexes not tailored to spatial data, such as the B+-Tree [134]. The alternative is to organize data to preserve spatial proximity information, which has been explored both in MonetDB [84] and PostreSQL [134] by using the Morton order.

The majority of the proposed solutions are traditional spatial index structures built in addition to the data model. Therefore, they require additional space resources which can introduce significant overhead, particularly for solutions based on the flat storage model [134]. An exception is the previously introduced Morton-replacedXY approach [84]. However, although the proposed solution integrates the Morton order into the flat model, it still requires significant space resources, as Morton codes and z values are stored per point cloud entry.

Part I

Data-Aware Spatial Joins

3 Joining Spatial Datasets with Contrasting Density

Many scientific and geographical applications rely on the efficient execution of spatial joins. Past research has produced several efficient spatial join approaches, however, the problem of efficiently joining two datasets with contrasting density, i.e., with the same spatial extent but with a wildly different number of spatial elements, has so far been overlooked. State-of-the-art data-oriented join approaches (e.g., based on the R-Tree) suffer from degraded performance due to overlap, whereas space-oriented approaches excessively read data from disk.

In this chapter we present GIPSY[1], a spatial join approach designed to efficiently join two datasets with contrasting densities. GIPSY uses the sparser dataset to guide the join process and therefore, it selectively retrieves and joins only the data needed. GIPSY relies on data-oriented partitioning to produce fine-grain partitions. At the same time, it avoids the problems associated with the overlap in the tree structure of the approaches based on data-oriented partitioning. Instead of traversing a tree structure top down, GIPSY traverses the data itself using a crawling approach. GIPSY is particularly efficient when joining a dense dataset with several sparse datasets.

3.1 Introduction

An increasing number of scientific or GIS applications depend on the efficient execution of spatial join operations. In geographical applications, for example, spatial joins are executed to determine the intersection or proximity between geographical features [133], i.e., landmarks, roads, etc. Medical imaging applications need an efficient spatial join to determine the proximity between cancerous cells [136] and in neuroscience the join is performed to find the intersection of neuron branches [90].

Many efficient approaches for disk-based spatial joins [18, 98] have been developed in the past. Unfortunately none of them can efficiently and scalably join two spatial datasets of substantially different density, i.e., of similar spatial extent but with a vastly different number

[1] GIPSY originally appeared in [99].

of spatial elements. Doing so, however, is important for several applications: it is needed to efficiently add a small number of roads or few elements to GIS datasets, to add the branches of one neuron to a spatial model of the neocortex and many other applications. The efficiency of the join is pivotal as it is oftentimes executed repeatedly to join several sparse datasets with one dense dataset.

To define the problem more formally, our goal is to develop an approach for repeated spatial joins of sparse datasets with one dense dataset. Given several sparse datasets A_i and a dense dataset B where $A_i << B$ (i.e., their spatial extent is similar, but the number of elements differ), the approach finds all pairs of spatial elements $a_k \in A_i$ and $b \in B$ so that a_k and b intersect. While any previously developed method [58] can be used to join a dataset A_i (with few elements) and B (with a massive number of elements), the state of the art is inefficient, as we will show with motivating experiments.

With the sparse datasets A_i repeatedly joined with the dense dataset B, building an index on B or on all A_i and B will speed up the join operation. The fundamental problem of existing approaches, however, is that with a very small A_i, only a small subset of B needs to be retrieved (and tested against A_i). Existing approaches based on space-oriented partitioning (e.g., PBSM [98]) create coarse-grained partitions and consequently the entire dataset B needs to be read for a join, leading to excessive disk accesses. Approaches based on data-oriented partitioning allow for a more fine-grained partitioning of the data, but require hierarchical trees (e.g., the synchronized R-Tree [18], indexed nested loop on the R-Tree [24]) to access the data and thus suffer from well documented problems like overlap and dead space, also resulting in excessive disk accesses.

We propose GIPSY, a novel approach that uses fine-grained data-oriented partitioning and thereby enables the join to read from B only the small subset needed. It avoids the overlap inherent in data-oriented partitioning by using an efficient crawling technique [97, 127] which is also used for range queries on spatial data. With this novel combination of crawling with data-oriented partitioning, GIPSY achieves a 2 to 18× speedup compared to the fastest approaches like the indexed nested loop [24] and PBSM [98] when joining several A_i with B.

The remainder of this chapter is structured as follows. We first motivate our work with an example application from neuroscience in Section 3.2. With an initial set of measurements we also demonstrate the shortcomings of the state of the art. In Section 3.3, we then explain our approach, GIPSY, and evaluate it in Section 3.4. We conclude in Section 3.5.

3.2 Motivation

GIPSY is motivated by the data management challenges the neuroscientists we collaborate with in the context of the Blue Brain Project (BBP [81]) face. We first describe the BBP, the spatial join challenges faced in it and then motivate the need for a new approach with an experimental analysis.

3.2.1 Blue Brain Project

In order to simulate and understand the brain, the neuroscientists in the BBP build the most detailed and biorealistic models with data acquired in anatomical research on the cortex of the rat brain. They have started to build small models of the elementary building block of the rat neocortex, a neocortical column of about 10,000 neurons. The structurally accurate microcircuits (or models) are built on massively parallel systems (currently the BlueGene/P with 16K cores). A visualization of a small microcircuit of a few thousand neurons is shown in Figure 3.1 (right).

The process of building the models starts with analyzing the neurons in the real rat brain tissue in the wet lab, measuring their electrophysiological properties as well as their morphology, i.e., their shape. As Figure 3.1 (left) shows, the morphology of a neuron is approximated with cylinders modelling the dendrite and axon branches in three dimensions.

To build a small scale model, several hundred or thousand neuron morphologies are put together in a spatial model. Before the model can be simulated, synapses (the places where electric impulses can leap over between different neurons) need to be placed. Prior research [66] has shown that an accurate model can be built by placing the synapses where the branches (or the cylinders representing them) of different neurons intersect. More precisely, synapses are placed where a cylinder representing an axon branch and a cylinder representing a dendrite branch intersect. The process of placing synapses thus equals to a spatial join of the axon and dendrite cylinders of the neurons.

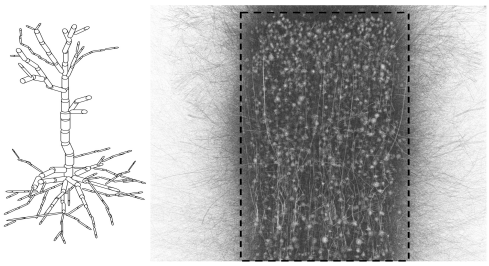
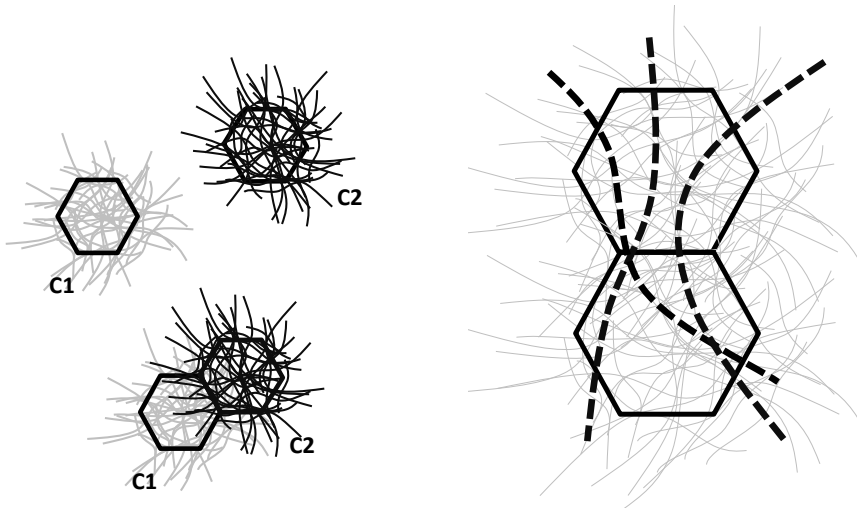


Figure 3.1 – Schema of a neuron's morphology modelled with cylinders (left) and a visualization of a model microcircuit comprised of thousands of neurons (right).

The models currently built and simulated in the BBP contain up to 500,000 neurons with the goal to increase the size of the models many times to first simulate the brain of the rat and ultimately the human brain with ~10^{11} neurons. More importantly, the circuits will become more detailed by modelling neurons (e.g., synapses and neurotransmitter) at the subcellular level and therefore packing orders of magnitude more spatial elements in the same space. Given that the spatial join is at the core of the model building its efficiency is pivotal.



(a) Combining Columns (b) Mesocircuits

Figure 3.2 – Illustration of the use cases.

3.2.2 Use Cases

Currently the BlueGene/P is used to perform the spatial join on a model. The model is partitioned and loaded into the 16K cores of it and each core will perform the spatial join and then report the result. Because the memory is limited to 1GB per core, the biggest model that can be built is a column, the smallest building block of the brain featuring about 10 million neurons. To attain the ultimate goal of simulating the entire brain, bigger models need to be built. The only way of doing so is to combine, on the disk of a single machine (or in a cluster), several columns into one big model, either by (1) combining several columns into one model or (2) connecting two columns with long ranging branches.

Combining Columns. Using the BlueGene/P to build the models limits the size of the biggest model to the size of the supercomputer's main memory, i.e., 10 million neurons or a column. To build bigger models, several columns need to be combined and hence the columns need to be spatially joined with each other. To speed up the join, only the branches (cylinders) penetrating the neighboring column are used for the join. Figure 3.2 (left) illustrates how columns C1 and C2 (view from top) are combined: only the few neuron branches (black lines) from C2 penetrating the neighboring column C1 (black lines inside C1) need to be joined with the neurons in C1 (gray lines in C1). All neurons and branches are modelled with thousands cylinders each. Therefore, the sparse dataset (the cylinders making up the black lines from C2 inside C1) containing several hundred thousand cylinders is joined with the dense dataset (the cylinders representing the gray lines of C1) having several hundred million cylinders.

Building Mesocircuits. In this use case one or few neuron branches are added to one or several columns. The added branches model the growth of mid-range fibers, i.e., model how branches penetrate a column. Also, in this case, the added branches interact with the neurons

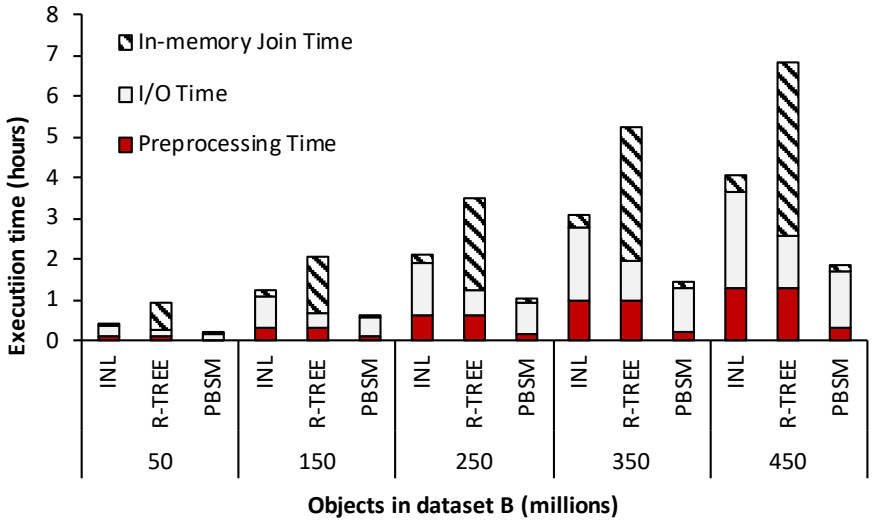


Figure 3.3 – Total execution time as a result of joining uniform datasets of different densities.

in the column, making the detection of touches between incoming branches and the rest of the circuit via a spatial join necessary. The number of cylinders added to the column in this case is typically several thousand. The sparse dataset is thus much smaller than in the previous use case, while the dense dataset contains a similar number of cylinders. Figure 3.2 (right) shows how the branches (dashed lines) are added to the neurons of the two columns (hexagons) to connect them.

3.2.3 Motivating Experiments

What is common among the aforementioned use cases is that two datasets of entirely different density, i.e., number of elements in the same space, are spatially joined. We illustrate the shortcomings of the current approaches when joining datasets with contrasting density in experimental analyses. We join a sparse dataset containing 800'000 elements with increasingly dense datasets containing between 50 and 450 million elements. We increase the density of the dense dataset to emulate increasingly detailed models (more elements in the same space). The experimental setup is described in more detail in Section 3.4.1.

We consider the partition based spatial-merge join (PBSM [98]), the synchronized R-Tree traversal (R-TREE [18]) and the indexed nested loop (INL [24]) approaches. The results of the join are shown in Figure 3.3. We distinguish between three phases: (1) *Preprocessing*, the time to index (or partition the data in case of PBSM), (2) *I/O Time*, the time to read partitions into memory, and (3) *In-memory Join Time*, time to join the partitions in memory.

The major problem for the data-oriented approaches, INL and R-TREE, is overlap between nodes in data structures [43], which increases alongside the increase in density. Overlap

leads to an increase in the number of I/Os and consequently, in the number of unnecessary comparisons (intersection tests). In the case of R-TREE, the overlap problem is more evident and affects primarily the in-memory join (it also affects I/O, but not to the same extent as the OS caches disk pages). The R-TREE approach is affected by overlap in the two R-Trees. Overlap at higher levels in each tree means that more R-Tree nodes overlap between the trees. Consequently, their children are compared pairwise, leading to a considerably bigger number of comparisons, which ultimately results in an increase of the in-memory join time.

The space-oriented approach, PBSM, on the other hand, suffers from the coarse-grain partitioning and random reads. Configuring PBSM is difficult: using a coarse-grain configuration produces bigger partitions and consequently, it significantly increases the number of unnecessary comparisons. On the other hand, using a very fine-grain configuration leads to excessive object replication – increasing the number of comparisons and the amount of data stored and transferred. The configuration used (25^3 partitions) in the experiment is identified with a parameter sweep. However, even though this configuration provides the best performance in terms of the total execution time, the produced granularity is not sufficient to filter a considerable amount of data from the dense dataset. In addition, being based on space-oriented partitioning, disk accesses to read the partitions, are mostly slow random reads, resulting in substantial I/O time.

3.3 The GIPSY Approach

With GIPSY we want to overcome the problems of approaches based on data-oriented as well as space-oriented partitioning. As outlined previously, space-oriented approaches cannot partition the dense dataset fine-grained enough so that the join can only retrieve the data needed from disk. Partitioning more fine-grained results in small partitions, therefore more replication and consequently a slower join. The approaches based on data-oriented partitioning, on the other hand, suffer from overlap resulting in unnecessary pages retrieved from disk followed by unnecessary comparisons.

3.3.1 Overview

The novelty of GIPSY lies in avoiding the coarse-grained partitioning of space-oriented approaches by using the fine-grained data-oriented partitioning. At the same time, it avoids the problems associated with the overlap in the tree structure of data-oriented approaches. Instead of traversing a tree structure top down, GIPSY cleverly traverses the data itself using a crawling approach [97, 127].

More precisely, GIPSY indexes the dense dataset and takes the elements of the sparse dataset and visits them one after the other by *walking* between them using the index on the dense dataset. Once it arrives at the location of a particular element e of the sparse dataset, it uses *crawling* to detect all elements of the dense dataset that intersect with e and then walks to the

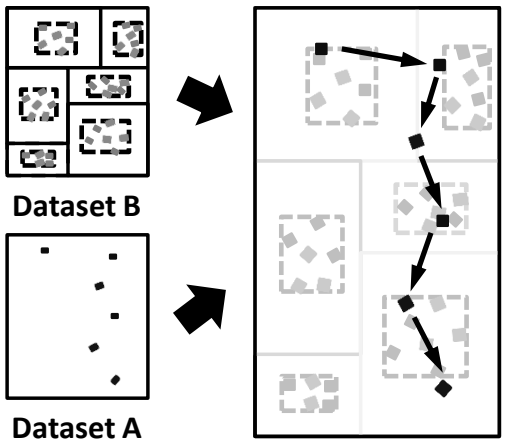


Figure 3.4 – GIPSY uses the sparse dataset to walk/crawl through the dense dataset.

next element. Using an index on the dense dataset and traversing it directed by the elements of the sparse dataset makes the join particularly efficient for the repeated join of a dense dataset with multiple sparse ones. Figure 3.4 illustrates how GIPSY uses the sparse dataset to direct walking in the dense dataset.

To enable GIPSY's novel approach of combining data-oriented partitioning with crawling to execute a spatial join, we need an efficient method to partition the dataset data-oriented, and to add & store the information needed for walking/crawling. Additionally, we need an effective method to find a start element for the walk as well as an order in which the elements of the sparse dataset can be visited with minimal distance between them.

In the following we discuss the methods, algorithms and data structures needed.

3.3.2 Indexing the Dense Dataset

Unlike mesh datasets indexed with DLS [97], the datasets we use (spatial datasets in general) do not have any inherent connectivity information like mesh edges. In GIPSY, we therefore first partition the dataset and then store information needed for crawling in additional data structures, similar to other crawling approaches [127].

Partitioning the Dense Dataset. To partition the dense dataset we use an approach similar to sort-tile-recursive (STR) [70], a method initially designed for bulkloading R-Trees. While we are not interested in the R-Tree it produces, its approach to data-oriented partitioning is useful so that spatially close elements can be stored on the same disk page, thereby preserving spatial locality. Similar to STR, GIPSY first sorts the dense dataset on the x-dimension of the element center and partitions the elements along this dimension. All resulting partitions are

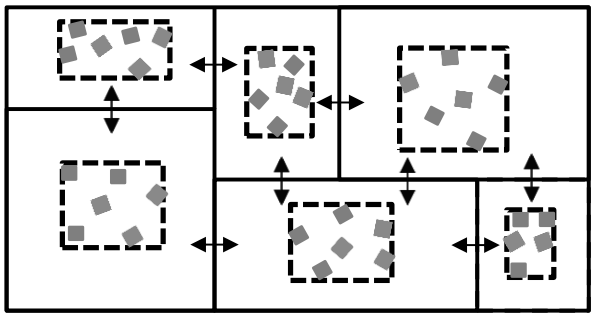


Figure 3.5 – Partioning of the dataset with solid lines for the partitions and dashed lines for the elements MBRs.

sorted on the y-dimension and partitioned again. Finally, the resulting partitions are also sorted on the z-dimension and partitioned, producing the final partitions.

Figure 3.5 illustrates the partitioning. The solid lines represent the partition MBRs (minimum bounding rectangle), whereas the dashed lines represent the elements MBRs that wrap more tightly the actual spatial elements. By choosing the size of the partitions at every step of the partitioning process, we can precisely determine the size of the final partitions. This (a) ensures that a partition fits on a disk page and (b) gives us a parameter to control the granularity of the partitioning. In GIPSY, the size of the partitions is always chosen so that it fits on a disk page, i.e., of size 4K or a multiple. GIPSY stores the elements in each partition together on a disk page called elements page.

Crawling Information. In order for GIPSY to work, we need to determine and store the information that enables walking/crawling. The relevant information are the partitions and their neighborhood relation, i.e., what partition neighbors other partitions.

Storing the Crawling Information: For each partition we store the minimum bounding rectangle (MBR) of the elements and the partition. The elements MBR is the minimum bounding box containing all elements on a page whereas the partition MBR is the minimum bounding rectangle of the partition. Most importantly, we also need to store the neighbors of each partition. We store all information in *summary records*: each record summarizes a partition p and stores a pointer to the elements page of p, p's partition MBR, p's elements MBR, as well as the neighbors of p (the partitions intersecting with p or touching p).

Determining the Crawling Information: The information of the partitions follows directly from the partitioning process. We determine the neighbors by performing a spatial self-join on the partition MBRs. This computes, for each partition p, what partitions n neighbor (touch or intersect) p. Any spatial join method can be used for the self join. Nevertheless, in GIPSY we use PBSM because we identified it as the quickest method to perform a one-off spatial join when both datasets have similar density.

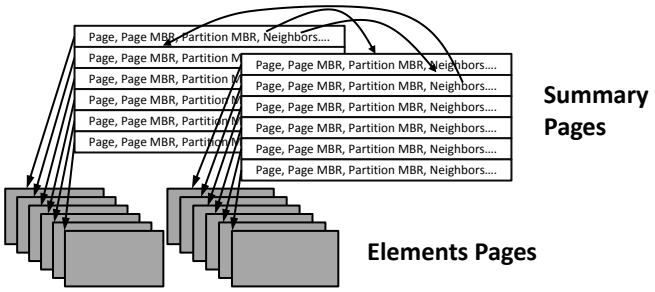


Figure 3.6 – The data structures of GIPSY: summary pages, elements pages and pointers between them (arrows between summary records).

As Figure 3.6 shows, all summary records are stored on disk pages called summary pages. We store as many summary records on a summary page as possible. When retrieving a summary record, it is likely that neighboring ones (spatially close ones) will also be retrieved and preserving spatial locality will thus improve performance. As a consequence, we use the Hilbert space filling curve [59], calculate the Hilbert value of each summary record (of the center of its partition MBR) and store on the same summary page summary records with consecutive Hilbert values. On the summary pages we do not store the partition identifier as a neighbor but instead store the identifier of the summary page it is stored. Such an approach simplifies and speeds up the join process as no mapping (correlating the summary page identifier with the disk page identifier) needs to be queried repeatedly.

3.3.3 Joining the Datasets

To finally join the datasets, GIPSY takes a sparse dataset A_i (without indexing it) and iterates over all its elements $a \in A_i$. It uses the start summary record at the beginning of the join and walks in the dense dataset to find the spatial location of the first element a_1 of the sparse dataset. For that matter it uses a directed walk: it recursively reads all neighboring summary records and picks the one closest to a_1 (smallest distance of the elements MBR to a_1). As Algorithm 1 illustrates with pseudocode, this process is repeated until a summary record intersecting with a_1 is found. If no neighbor record closer to a_1 can be found and the elements MBR of the closest record still does not intersect with a_1, then a_1 does not intersect with any element from B.

Once an *intersection record*, a summary record of which the elements MBR intersects with the element a_1, is found the directed walk ends and the crawl phase starts. The goal of the crawl phase is to find all elements of the dense dataset intersecting with a_1. Starting with the *intersection record*, the crawl phase, similarly to the walk phase, recursively visits all neighbors until no more element intersecting with a_1 can be found. More precisely, it starts with the *intersection record* and recursively retrieves all summary pages that contain referenced neighbor records. If the elements MBR of a summary record intersects with a_1,

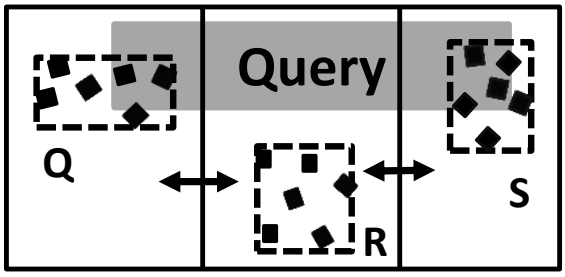


Figure 3.7 – Starting with partition Q, GIPSY has to recursively visit all neighbors with intersecting partition MBRs.

Algorithm 1: Directed Walk

Input: startR: start crawl record
a_k: spatial element of sparse dataset A_i
Output: closestR: closest summary record to a_i

$closestR = startR$;

while $(distance(closestR.elMBR, a_k) > 0\ AND\ !chekgettingaway())$ **do**
  $records$ = read all neighbor records of $closestR$;
  foreach *summary record* $r \in records$ **do**
    if $distance(r.elMBR, a_k) < distance(closestR.elMBR, a_k)$ **then**
      $closestR = r$;
    end
  end
end
return *closestR*

then the elements page is retrieved and all elements are tested for intersection. If the partition MBR of a summary record does not intersect, then the neighbors are not visited and hence the crawl phase ends when no more crawl record with a partition MBR intersecting with a_1 can be found. Algorithm 2 illustrates the crawl phase with pseudocode. If no summary record of which the elements MBR intersects with a_1 can be found, then a_1 does not intersect with any element and we walk to the next element in A.

The algorithm also illustrates why GIPSY needs to store and use both, the partition and elements MBRs. The elements MBR is needed in the join process to determine whether or not to retrieve an elements page (if the query intersects with the elements MBR). The partition MBR, on the other hand, is needed to guarantee correctness: given a partition (and its summary record) Q, even if the elements MBR of Q's neighbor R does not intersect with the query, R's neighbor S elements MBR might. Consequently GIPSY cannot stop visiting R's neighbors only because its elements MBR does not intersect with the query, but only if the partition MBR does not intersect, as Figure 3.7 illustrates.

Algorithm 2: Crawl Algorithm

Input: intersectionR: crawl record with page MBR intersecting with $a \in A_i$;
range: MBR of spatial element $a \in A_i$
Output: result: spatial elements
Data: squeue: summary record queue
visitedqueue: already visited elements queue

```
enqueue intersectionR into squeue
while squeue ≠ ∅ do
    dequeue summary record s from squeue
    if s.elementsMBR intersects with range then
        retrieve elements page ep referenced in s
        foreach element ∈ p do
            if element MBR intersects range then
                put element into result
            end
        end
    end
    if s.partitionMBR intersects with range then
        foreach neighbor in s do
            if neighbor is not in visitedqueue then
                enqueue neighbor summary record in squeue
            end
        end
    end
end
return result
```

As the pseudocode in Algorithm 3 shows (using Algorithm 1 as directedWalk and Algorithm 2 as crawl), after finding all elements of the dense dataset that intersect with a_1, the same process, i.e., directed walk and then crawling, is repeated to find the intersections of the following elements a_k until all elements of A_i intersecting with elements of the dense dataset B are identified.

3.3.4 Visiting Order

The order in which the elements of the sparse dataset are visited has an impact on the distance walked and consequently also on the execution time of the join. While walking a longer distance and retrieving more summary pages does not automatically need to translate into more time needed to access the disk (due to caching of the OS), walking longer will, however, means more time is spent on comparing summary records to elements of the sparse dataset.

An ideal visiting order minimizes the overall distance walked, similar to the travelling salesman

Algorithm 3: GIPSY Join

Input: elements: array of spatial elements A_i
startR: summary record from where to start join
Output: elements: set of elements A_i intersecting with element from B
Data: intersectionR: summary record holding the current intersection record
intersectingE: elements ∈ B intersecting with an element inA_i

$intersectionR = startR$;

foreach $elementa \in elements$ **do**
    $intersectionR = directedWalk(a, intersectionR)$;
    $intersectingE = crawl(intersectionR)$;
    add all $intersectingE$ to $elements$;
end
return *elements*

problem (TSP). Unfortunately, the TSP is NP-hard and we have to resort to heuristics to find an order that approximates the optimal order in reasonable time. We have implemented several strategies to sort the sparse dataset and evaluate them in the experimental section in Section 3.4.

3.3.5 Start Point

To visit the elements of the sparse dataset, GIPSY needs to start at a particular summary record of the dense dataset and walk through it. GIPSY could start with a random (or chosen by some heuristic) summary record, use a directed walk to the closest summary record of a_1 (the first element of the sparse dataset) and then start the join process. This method, however, depends on the randomly chosen summary record as well as the sparse dataset and may thus involve an infeasibly long walk.

To reduce the distance between the start point and the first element of the sparse dataset, we index all summary records of the dense dataset. Any spatial index could be used to index the dense dataset so that a first summary record close to an element of the sparse dataset can be retrieved. To avoid the issue of overlap and also to speed up the process of building the index we refrain from using an R-TREE or related spatial indexes. Instead we calculate the Hilbert value of each summary record (the Hilbert value of the center of the elements MBR) and index them with a B+-Tree.

To find the summary record to start from, we execute the range queries (the Hilbert values of the sparse dataset elements) on the B+-Tree in order to find the first intersection, i.e., the summary record with the closest Hilbert value to one of the elements of the sparse dataset. This summary record does not necessarily contain the first element of the sparse dataset but will be spatially close to it and GIPSY will walk to it and start the traversal there.

The B+-Tree can also be reused in case of an extremely sparse dataset: instead of an infeasibly long walk between two elements a_i and a_{i+1} of the sparse dataset, it may be more efficient to use the B+-Tree instead to find a summary record close to a_{i+1}. In our experiments, however, we have not encountered a dataset where using the B+-Tree repeatedly improves performance.

3.4 Experimental Evaluation

In this section we describe the experimental setup & methodology, compare GIPSY against state-of-the-art spatial join approaches and analyze its performance.

3.4.1 Setup

Hardware. The experiments are run on a Red Hat 6.3 machine, with 2 quad-coe AMD Opteron CPUs at 2700 MHz, with 4 GB RAM and 4 SAS disks of 300GB (10000 RPM) capacity as storage. We use one of the disks for the experiments, i.e., no RAID configuration is used.

Software. All algorithms are implemented single-threaded in C++ for a fair comparison.

Setting. We compare the indexed nested loop join (INL), synchronized R-Tree traversal (R-TREE), partition based spatial-merged join (PBSM) and our approach GIPSY. R-TREE and PBSM use the plane sweep algorithm as the in-memory join.

Due to the absence of appropriate heuristics, we set the parameters of related approaches optimally after a parameter sweep. In case of PBSM we found the configuration with 25^3 partitions to be the most efficient. This configuration provides the best trade-off between the number of elements needed to be compared by the plane sweep algorithm and the number of elements replicated, deduplicated, additionally written/read to/from disk. INL and R-TREE have shown the best performance with a fanout of 135. The disk page size in all experiments is 8 KB. Experimental conditions assume a cold file system cache, i.e., after the preprocessing/indexing step OS caches and disk buffers are all cleared.

3.4.2 Experimental Methodology

We use two different types of datasets in the experiments: (1) to control the dataset characteristics (number, size and distribution of elements) and demonstrate general applicability we use synthetic datasets and (2) to demonstrate the impact on our use cases we also use neuroscience datasets.

Synthetic Datasets. We create synthetic datasets by distributing spatial boxes in a space of 1000 space units in each dimension of the three-dimensional space. The length of each side of each box is determined by a uniform random distribution between 0 and 1. Spatial elements are distributed in space depending on the data distribution.

We use three different data distributions - uniform, normal ($\mu = 0$, $\sigma = 220$) and clustered, and always join datasets of the same distribution. For the clustered dataset, we choose uniformly randomly centers of the clusters in the three-dimensional space and place between 500 to 1000 spatial elements around the cluster center using a normal distribution ($\mu = 0$, $\sigma = 220$). The number of spatial elements in the datasets is between 10K and 450M. The corresponding size on the disk is between 468KB and 20GB.

Neuroscience Datasets. To evaluate GIPSY on real data we use a small part of the rat brain model represented with 450 million cylinders as elements. We take from this model a contiguous subset with a volume of 285 μm^3 and approximate the cylinders with minimum bounding boxes. In the spatial join process, axons are represented by one dataset, dendrites by the other, and the detected intersections represent synapses. The number of elements in the datasets is between 10K to 250M. The corresponding size on the disk is between 468KB and 11GB.

Approach. Spatial joins typically involve two steps: filtering followed by refinement. The filtering step finds pairs of spatial elements whose approximations intersect with each other, while the refinement step detects the intersection between the actual shape of the elements. Considering these two steps are independent in terms of their implementation and the refinement step is application specific, we focus on the filtering step.

In all experiments we fix the size of dataset A (sparse dataset) and gradually increase the size of dataset B (dense dataset). We increase the density of the dense dataset to emulate the increasingly detailed models the neuroscientists build. To build more biorealistic models, they increase the number of elements in the same space, i.e., they increase the density, leading to growing overlaps in indexes based on data-oriented partitioning.

In all experiments we measure the total execution time and the number of I/O operations during the spatial join process. The total execution time is measured for two different scenarios, for a one-time operation (joining dataset A with B) and for a repeated spatial join operation (joining several datasets A_i with one B).

We break the total execution time in preprocessing time, I/O time and in-memory join time. The preprocessing time is the time necessary to build the initial data structures (PBSM: partition creation, element assignment; R-Tree based approaches: index building; GIPSY: space partitioning, neighborhood information introduction, B+-Tree building). The I/O time is time spent on data loading during the join process and the in-memory join time is the time needed to join data in memory, i.e., comparing the spatial elements and related operations.

In the case of the repeated spatial join, we reuse the index (partitions for PBSM, tree for R-TREE and INL) created during the first spatial join process. The total execution time for these experiments thus contains the preprocessing time only once.

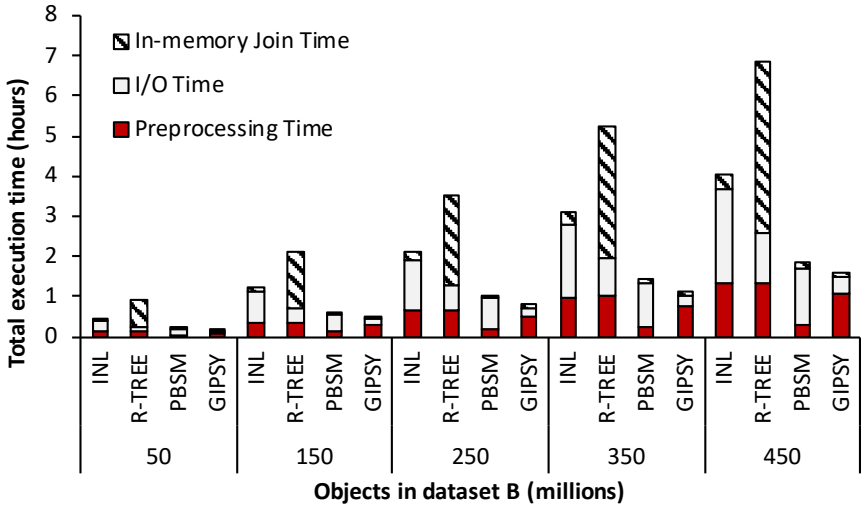


Figure 3.8 – Total execution time as a result of one spatial join, combining columns.

3.4.3 Combining Columns

One-time Join. We evaluate GIPSY on synthetic data with two sets of experiments, both inspired by the two neuroscience use cases described in Section 3.2.2. This set of experiments is designed according to the *combining columns* use case. More precisely, in the following experiments we fix the size of dataset A to 800K elements and join it once with datasets B of increasing size from 50M to 450M, in steps of 100M. All datasets have a uniform distribution. Figure 3.8 shows the total execution time broken down into preprocessing time, I/O time, and in-memory join time. In this experiment GIPSY outperforms all other algorithms and its improvement over PBSM, the fastest state-of-the-art approach, is between 16% - 25%.

The performance of state-of-the-art data-oriented approaches, i.e., R-tree based approaches, is degraded due to overlap. In the case of R-TREE, the overlap problem is more evident and affects primarily the in-memory join (it also affects I/O, but this is not obvious because the OS caches disk pages). The R-TREE approach is affected by overlap in the two R-Trees. Overlap at higher levels in each tree means that more R-Tree nodes overlap between the trees and consequently their children are compared pairwise, leading to a considerably bigger number of comparisons which ultimately results in an increase of the in-memory join time.

INL essentially repeatedly executes a small range query on the R-Tree for every element in A. The in-memory join time hence does not grow considerably with increasing overlap, because every inner node of the R-Tree retrieved during query execution only has to be compared against the range query (unlike R-TREE where due to overlap all children of overlapping nodes need to be compared with each other). Overlap in INL, however, means that more nodes and thus disk pages need to be read (in a tree without overlap, a range query can be executed by accessing as many nodes as the tree is high).

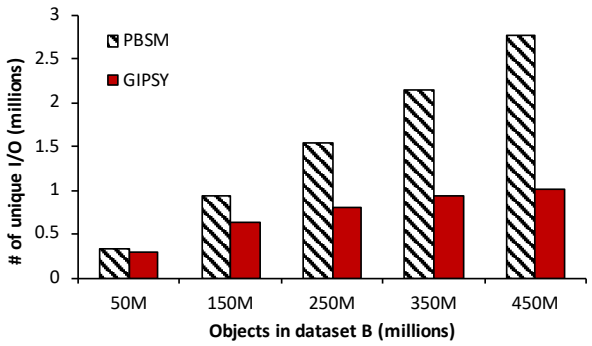


Figure 3.9 – Number of I/Os as a result of one spatial join, combining columns.

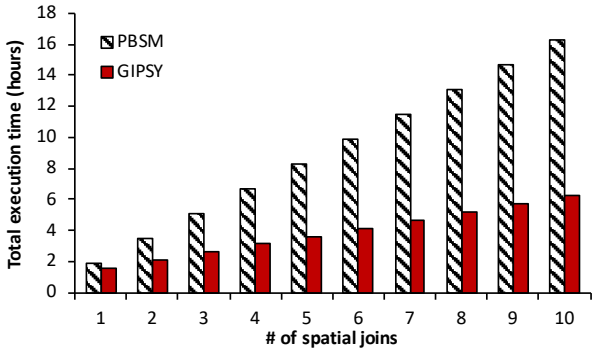


Figure 3.10 – Total execution time as a result of repeated join, combining columns.

PBSM has a low cost partitioning strategy since it is based on simple, space-oriented partitioning. Likewise, as opposed to data-oriented partitioning, it does not invest additional processing resources to ensure sequential access to data during the join phase. Overall, PBSM is the fastest state-of-the-art approach, but being based on a space-oriented partitioning, it has to retrieve most of the data. In addition, disk accesses to read the partitions are mostly slow random reads. The I/O time of PBSM consequently makes up most of the execution time.

In comparison to other algorithms, GIPSY spends significantly less time on the in-memory join (i.e., performing comparisons) and I/O operations. On average, 60% of the total execution time is spent on the preprocessing step.

In the remaining experiments we primarily compare GIPSY with PBSM, the fastest state-of-the-art approach. As the previous experiments show, the main problem of PBSM is unnecessary data retrieval when joining datasets of different densities. In the next experiment, we therefore measure the unique disk pages read during the spatial join when we increase the density of the dense dataset B. As Figure 3.9 shows, with increasing density of B, also the I/O ratio between PBSM and GIPSY increases. For instance, in the case of joining uniform datasets of 800K with

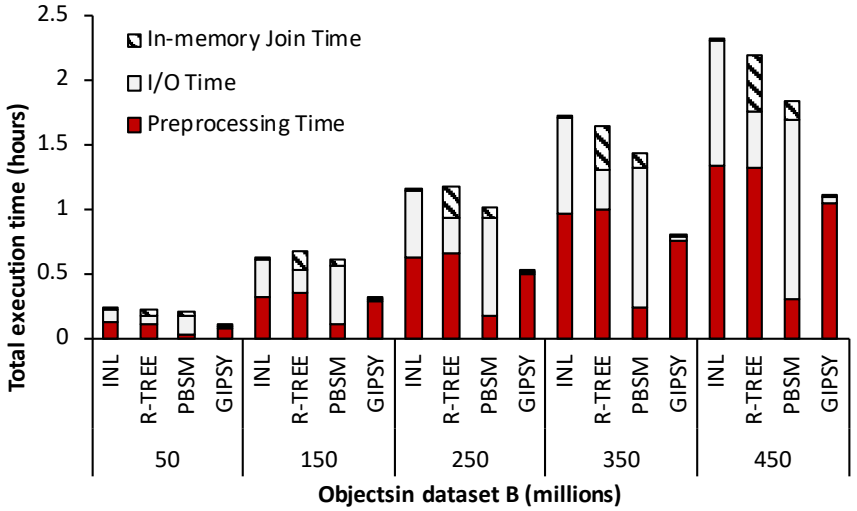


Figure 3.11 – Total execution time as a result of one spatial join, building mesocircuits.

450M elements, PBSM needs to read 2.71x more unique disk pages than GIPSY. GIPSY reduces the data read from disk by using fine-grain data-oriented partitioning.

Repeated Joins. In the next experiment we join 10 different sparse datasets of 800K elements with the same dense dataset containing 450M spatial elements. The total execution time is the sum of all spatial joins and includes the time for building the index on the dense dataset B just once (created during the first spatial join and reused thereafter). As Figure 3.10 shows, for repeated joins, after 10 spatial joins, GIPSY already outperforms PBSM by a factor of 2.62.

3.4.4 Building Mesocircuits

One-time Join. Inspired by the *building mesocircuits* use case described in Section 3.2.2, in this set of experiments we join one (or several) very sparse datasets with a dense dataset. We use the same experimental methodology as in the previous set of experiments. However, we decrease the size of the sparse dataset by a factor of 800, i.e., it contains 10K spatial elements.

The results of these experiments are shown in Figure 3.11. Compared to the previous experiments, only INL differs in relative performance, as its execution is slower compared to the R-TREE. The relative performance of the R-Tree-based approaches, however, has improved and if we take into account index reuse, i.e., we do not consider the preprocessing time, PBSM is now slower than the rest of the algorithms.

R-Tree-based approaches perform better than PBSM (in the case of index reuse) because they are essentially a compromise between (data-oriented) fine-grain partitioning and the overlap problem. Given that one of the datasets is significantly sparser, R-TREE is not affected by the overlap to the same extent. Similarly, the number of R-Tree traversals decreases for INL.

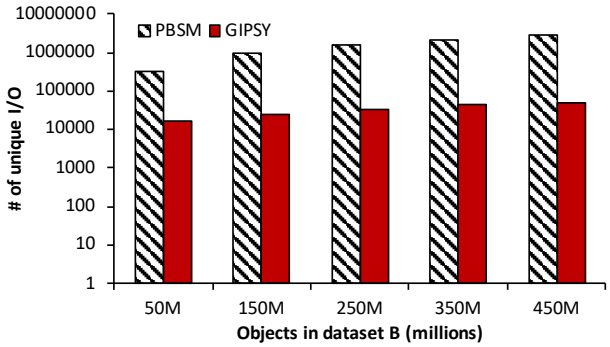


Figure 3.12 – Number of I/Os (logscale) as a result of one spatial join, building mesocircuits.

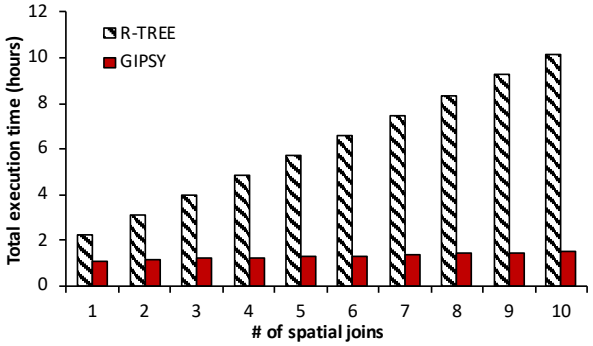


Figure 3.13 – Total execution time as a result of repeated join, building mesocircuits.

Consequently, the in-memory join phase is significantly reduced for R-TREE and INL. PBSM, on the other hand, still needs to retrieve all of the dense dataset. Figure 3.12 illustrates the total number of I/O operations (logscale) executed during the join process of PBSM and GIPSY (without considering the preprocessing phase).

Based on the total execution time, GIPSY achieves the best results with an average improvement of $2x$ compared to R-TREE. However, during a single spatial join GIPSY spends on average 90% of total execution time in the preprocessing phase building the index. Not considering the preproceessing phase, i.e., if the index is reused for repeated joins, GIPSY achieves a total speedup up to $17.95x$ compared to R-TREE.

Repeated Joins. The experimental results comparing GIPSY with R-TREE when repeatedly joining datasets are shown in Figure 3.13. We join 10 different sparse datasets, each containing 10K spatial elements with one dense dataset of 450M elements. The total execution time is the sum of all previous spatial joins and includes the time necessary for the preprocessing step only once. When joining all 10 sparse datasets with the dense dataset, GIPSY attains a speedup compared to the synchronized R-Tree traversal of $6.78x$. The total execution time, in the case of GIPSY, appears to be a flat line, as it increases minimally with each spatial join.

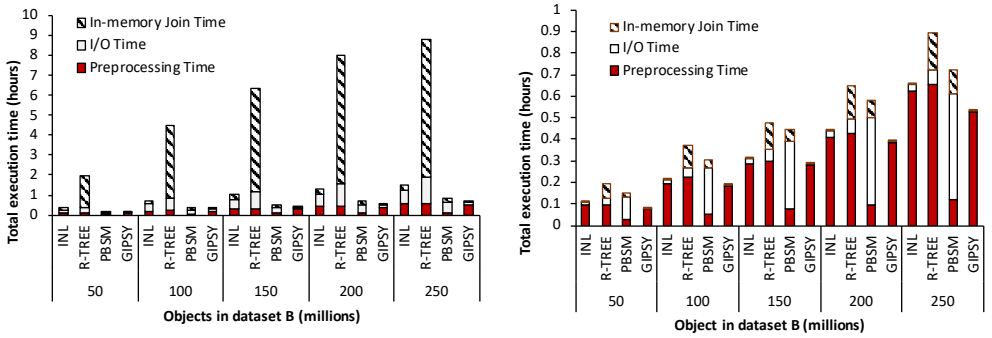


Figure 3.14 – Total execution time as a result of one spatial join for neuroscience datasets, combining columns (left) and building mesocircuits (right).

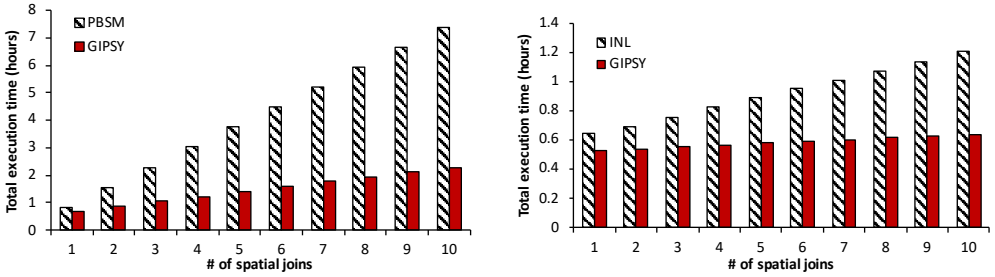


Figure 3.15 – Total execution time as a result of repeated join for neuroscience datasets, combining columns (left) and building mesocircuits (right).

3.4.5 Neuroscience Datasets

As a litmus test and to demonstrate the usefulness of GIPSY for the neuroscientists, we also test its performance on neuroscience datasets. We use a similar methodology as before and set the size of sparse dataset A to 450K spatial elements (dendrites) for the *combining columns* use case and to 10K for the *building mesocircuits* use case. The dense dataset B is increased from 50M to 250M (axons).

Figure 3.14 shows the results of one-time spatial join for both use cases. The R-TREE approach is significantly slower than in the previous experiments, as data distribution increases the overlap in both R-Trees. We additionally measure the number of inner node I/Os and notice a significant increase compared to the spatial join performed on the uniform datasets. The increase in the inner node I/Os goes from $2.34x$ (for the dataset of 50M elements) to $3.27x$ (for the dataset of 250M elements), which is a good indication of increased overlap.

In the case of the repeated spatial join, we compare GISPY to the second best approaches, i.e., PBSM (for the combining columns use case) and to INL (for the mesocircuit use case). The results in Figure 3.15 show a speedup of $3.5x$ compared to PBSM, and of $2x$ compared to INL.

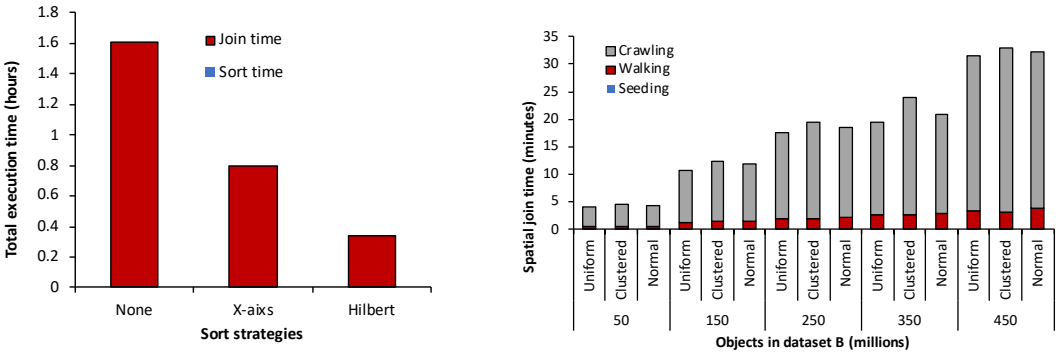


Figure 3.16 – Impact of sort strategies (left) and data distributions (right).

3.4.6 GIPSY Sensitivity Analysis

In the following we analyze the impact of the sorting strategy, page size and data distribution on the performance of GIPSY.

Impact of Visiting Order. In the walking phase, GIPSY walks through the dense dataset directed by the elements of the sparse dataset. The order in which the elements of the sparse dataset are visited should ensure that the overall distance is as small as possible. Otherwise, the number of I/Os and comparisons can increase significantly.

In the following, we evaluate the impact of different visiting strategies on the performance of GIPSY. We execute a spatial join between a sparse dataset with 800K spatial elements and a dense dataset with 100M elements. Both datasets have uniform distributions. We compare four different sort strategies on the sparse dataset: none (use dataset as it is), a nested loop sort, X-axis sort and Hilbert sort. The X-axis sort sorts the elements based on their x-coordinate, while the Hilbert sort sorts based on the Hilbert value [59] of the center of the element. The nested loop sort compares all elements pairwise and visits them in the order of minimal pairwise distance. In each step we exclude the elements for which we have already found the minimum distance from further consideration.

The results of this experiment are shown in Figure 3.16, where we divide the total execution time into sort and join time. Due to the long sort execution time, we exclude the nested loop sort from the experiment. The time necessary to sort the sparse dataset in case of X-axis and Hilbert sort is negligible. GIPSY's performance is degraded by a factor of $4.8x$ when not using any sort strategy and $2.4x$ when sorting on the x-dimension only. Finally, join based on Hilbert sort has the best performance.

Impact of Data Distribution. In the following experiments we measure the impact of data distribution on GIPSY by running the experiments from Section 3.4.3 on datasets with uniform, clustered and Gaussian distribution. Figure 3.16 illustrates the spatial join time. To analyze different stages in GIPSY, we breakdown its total execution time into: Seeding - time necessary to obtain start point, Walking - walk time and Crawling - the crawling phase.

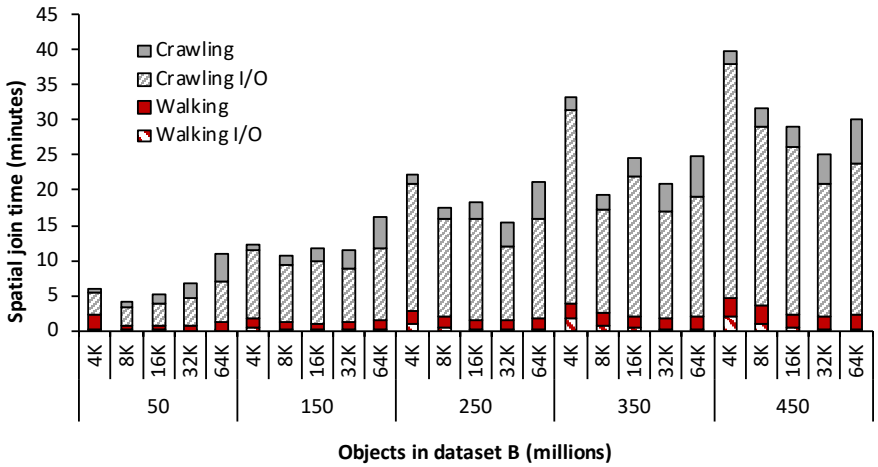


Figure 3.17 – Spatial join time, varying the page size from 4KB to 64 KB.

The overall spatial join time does not vary significantly for the three different distributions. GIPSY takes slightly more time for joining clustered data, followed by Gaussian and uniform data. As Figure 3.16 shows, GIPSY spends significantly less time on the directed walk compared to the crawling phase. This was the initial assumption for developing GIPSY - we rely on spatial element proximity and ensure a walk as small as possible, by following a particular visiting order. The crawling phase, on the other hand, depends heavily on the average number of neighbors (the denser the dataset is, the more neighbors we have to examine). The time needed to find a start point is insignificant in all cases.

Impact of Page Size. To measure the impact of page size on GIPSY performance, we execute the experiments from Section 3.4.3 (combining columns use case), varying the size of page from 4KB to 64KB.

Figure 3.17 shows the result of the experiment. GIPSY's join time is divided into: walking I/O - time spent on retrieving neighborhood information in the walking phase, walking - walking related operations (e.g., distance calculations), crawling I/O - time spent on I/O operations during crawling phase, and crawling - crawling related operations (e.g., overlap detection).

A change in the page size is a trade-off. Increasing the page size, on the one hand, leads to fewer neighbors per summary record and fewer random reads. At the same time, one node contains more data, i.e., more summary records that need to be examined. Because the element pages contain more elements, their page/partitions MBRs increase, and consequently we have more unnecessary comparisons. The results of the join between datasets with 800K and 450M elements for page sizes of 4KB and 64KB confirm our expectation: when using the bigger page size, I/O time decreases, while the time spent on crawling and walking related operations increases. In our experiments, the best performance for dense datasets is obtained for a page size of 32KB.

3.5 Conclusions

In this chapter we identify the problem of joining datasets of contrasting density, i.e., joining several sparse datasets with a dense dataset. State-of-the-art approaches do not join these datasets efficiently. Data-oriented approaches suffer from overlap resulting in excessive reads from disk and unnecessary comparisons. Space-oriented approaches cannot partition the datasets fine-grained enough and typically the entire dataset has to be read from disk, although only a small part is necessary.

The novelty of GIPSY, the approach we develop to tackle the challenge, lies in the efficient combination of crawling with data-oriented partitioning to join spatial datasets. GIPSY indexes the dense dataset with a data-oriented approach and avoids overlap through crawling: the sparse datasets are used to crawl through the index of the dense dataset. Only small parts of the dense dataset needed for the join are retrieved.

In our experiments we show the effectiveness of GIPSY, as it outperforms state-of-the-art disk-based spatial join algorithms between a factor of 2 & 18, when not considering the cost of preprocessing. It is particularly efficient when joining a dense dataset with several sparse datasets. We have tested GIPSY on neuroscience, but also on synthetic datasets, demonstrating that it can be efficiently used on spatial datasets from other domains/applications as well.

4 Adapting to Spatial Datasets Characteristics

Spatial joins are becoming increasingly ubiquitous in many applications, particularly in the scientific domain. While several approaches have been proposed for joining spatial datasets, each of them has a strength for a particular type of density ratio among the joined datasets. More generally, no single proposed method can efficiently join two spatial datasets in a *robust* manner with respect to their data distributions. Some approaches do well for datasets with contrasting densities while others do better with similar densities. None of them does well when the datasets have locally divergent data distributions.

In this chapter we present TRANSFORMERS[1], an efficient and robust spatial join approach that is indifferent to such variations of distribution among the joined data. TRANSFORMERS achieves this feat by departing from the state-of-the-art through adapting the join strategy and data layout to local density variations among the joined data. It employs a join method based on data-oriented partitioning when joining areas of substantially different local densities, whereas it uses big partitions (as in space-oriented partitioning) when the densities are similar, while seamlessly switching among these two strategies at runtime. We experimentally demonstrate that TRANSFORMERS outperforms state-of-the-art approaches by a factor of between 2 and 8.

4.1 Introduction

In many different applications, the efficient execution of spatial joins becomes increasingly important. In scientific applications, for example, spatial joins are used to determine the location of synapses in brain models [82], in medical imaging to determine proximity of cells and in geographical information systems spatial joins detect collisions between geographical features like houses, roads, etc.

Given the importance of the application, several methods have been developed to perform disk-based spatial joins [18, 98]. Methods developed in the past can efficiently join two or more

[1] TRANSFORMERS originally appeared in [100].

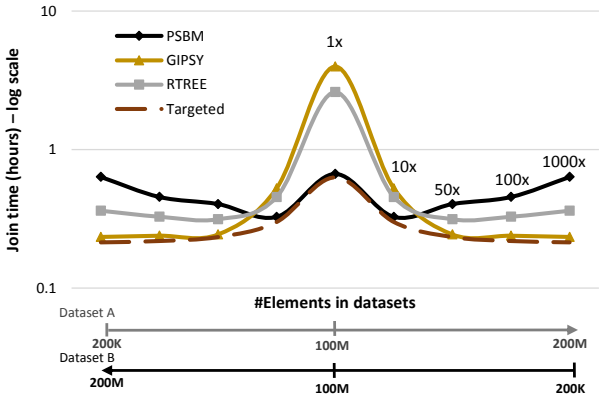


Figure 4.1 – Join time for datasets with variable relative density.

disk-based spatial datasets of uniform distribution of element locations. They are, however, inadequate when joining two spatial datasets, each with a skewed distribution of element location [99]. Formally, the goal is to develop a robust and efficient method to spatially join two disk-based datasets, each with a non-uniform distribution of element location, i.e., with *locally varying* densities. More precisely, each dataset D can have areas d_i with a considerable difference in density such that $\forall i, j$ with $i \neq j$ $|d_i| << |d_j|$ or $|d_i| >> |d_j|$ along with areas of similar density, $|d_i| \approx |d_j|$. When joining such datasets A and B we have to efficiently join areas $a_i \in A$ and $b_i \in B$ with similar spatial extent and location regardless of their density.

As we show with an experiment in Figure 4.1, no previously proposed approach achieves to join all combinations of density ratios of a_i and b_i, e.g., $|a_i| \approx |b_i|$, $|a_i| << |b_i|$ or $|b_i| << |a_i|$, in a *robust* manner. In this experiment we join different combinations of datasets A and B and measure the join time. More precisely, we steadily increase the density of dataset A (starting from 200K spatial elements) and decrease it for B (starting from 200M) and measure the execution time for joining each combination. The two datasets represent areas with different or similar density. The numbers above the curve indicate the ratio of density between the datasets; a more detailed explanation is given in Section 4.2.1.

As the experimental results in Figure 4.1 show, none of the existing approaches performs best in every situation. Approaches based on space-oriented partitioning (e.g., PBSM [98]) do well when joining datasets of similar density while data-oriented partitioning approaches (e.g., based on the R-Tree [18, 24] and particularly GIPSY [99] (Chapter 3)) are more efficient when joining datasets with contrasting density. None of the existing approaches joins the datasets (or areas) with robust performance across different density ratios.

In this chapter we thus introduce TRANSFORMERS, a novel disk-based join approach that handles this robustness problem with respect to local density variations and targets to achieve performance as is shown in Figure 4.1 (Targeted). For all areas $a_i \in A$ and $b_i \in B$, TRANSFORMERS decides *locally* which area is dense and which is sparse and adapts the join strategy as

well as the data layout accordingly. With its adaptive join strategy, TRANSFORMERS achieves a more robust join performance across different density ratios and outperforms previous work by a factor of between 2 and 8.

Our contributions are as follows:

- We show that static strategies in the join phase lead to sub-optimal, non-robust performance when joining datasets with non-uniform distributions.
- We develop TRANSFORMERS, a novel approach that detects local variations in distributions and adapts its strategy and the data layout on the fly accordingly.
- We demonstrate robustness as well as substantial performance improvements achieved with TRANSFORMERS on scientific and synthetic datasets.

The remainder of this chapter is organized as follows. In Section 4.2 we motivate TRANSFORMERS with an initial set of measurements confirming our assumptions. We give an overview of our approach in Section 4.3 and then discuss in detail the indexing process in Section 4.4, the join process in Section 4.5 as well as transformations in Section 4.6. In Section 4.7 we demonstrate the performance of TRANSFORMERS and draw conclusions in Section 4.8.

4.2 Motivation

Spatial joins have become a crucial operation across different scientific and business applications. Frequently the two datasets to be joined have a considerably different density and data distribution. For example, while dataset A has a uniform distribution and density, the join performance may be affected by the local variation in distribution and density of dataset B.

In Figure 4.2 we illustrate several examples of local variations in distribution and density. *Uniform* (left) illustrates two datasets with similar distribution and density throughout the area they cover. In case of *contrasting density* (middle) both datasets have similar distribution, however, skew is introduced through the different number of elements in the datasets. The datasets in *contrasting distribution* (right), on the other hand, have a similar number of elements but a different distribution. In the latter two cases, *contrasting density* and *contrasting distribution*, a join between two datasets will join areas with considerably different densities. As we illustrate with the following experiments, skew due to these variations in density leads to significant overhead, i.e., unnecessary processing.

4.2.1 Motivating Experiment

We illustrate the shortcoming of the state of the art with experiments where we join datasets with contrasting densities. To achieve an approximation of joining two disk-based datasets that differ significantly in local densities, we join nine pairs of datasets with uniform data distribution whose density ratio ($|a|/|b|$) varies between 10^{-3} and 1000 (numbers shown above the curves in Figure 4.1). To obtain nine pairs of datasets with contrasting densities, we

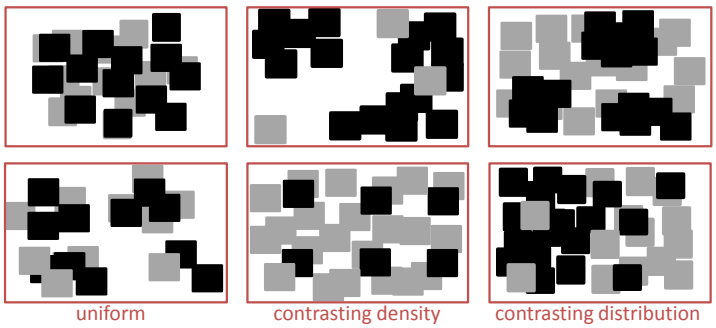


Figure 4.2 – Illustration of variations in distribution and density. Each figure shows two datasets, one with grey elements and the other with black ones.

increase the density of one dataset (starting from 200K elements) and decrease it for the other (starting from 200M) in consecutive steps.

We measure the execution time (without taking into account the indexing phase) for the join for the fastest and most broadly used disk-based spatial join methods, i.e., PBSM [98], the Synchronized R-Tree (R-Tree [18]) and GIPSY [99]. We use the best configuration for each approach, e.g., the number of partitions/tiles for PBSM, page size and fanout for the R-Tree. The results are shown in Figure 4.1.

When joining a sparse area $a_i \in A$ with a dense area $b_i \in B$, only a very small subset needs to be retrieved from b_i (and tested against a_i). Approaches based on space-oriented partitioning like PBSM [98], however, read considerably more data than is required from b_i and thus also require more comparisons. Due to coarse-grained partitioning inherent in these methods, almost all of b_i is read for the join, leading to excessive disk accesses and comparisons (points 1000x, 100x, 50x). Space-oriented partitioning methods, on the other hand, are efficient when joining areas of similar density (point 1x).

Data-oriented partitioning approaches (based on the R-Tree [43] or others, e.g., the synchronized R-Tree [18]) use a very fine-grained partitioning that enables them to retrieve data very selectively. As Figure 4.1 shows, doing so proves efficient on contrasting densities but their inherent problem of structural overlap leads them to read and test more data than necessary, making them comparatively slow when joining similar densities. GIPSY (Chapter 3) minimizes the impact of overlap by using the sparse dataset to selectively retrieve the data needed from the dense dataset, relying on connectivity information instead of a hierarchical tree traversal. By doing so, GIPSY efficiently executes a join between a sparse and a dense dataset; however, it is inefficient when joining datasets of similar density. The problem of GIPSY is that it, like other approaches, uses a static strategy and does not consider the characteristics of the datasets.

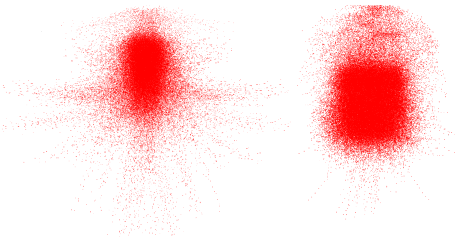


Figure 4.3 – Neurocience data: axons (left) and dendrites (right).

4.2.2 Motivating Application

To better understand the brain and develop new drugs for brain related diseases, the scientists in the Human Brain Project [82] build small-scale spatial models of the rat brain for brain simulations. The spatial models they design are based on millions of three-dimensional cylinders where several thousand cylinders together reconstruct the spatial shape of one neuron. To determine the locations of synapses they perform a disk-based spatial join between two types of neurons (or their corresponding cylinders), axons and dendrites. Wherever an axon intersects with a dendrite, a synapse is placed [66]. The amounts of data involved in the join make it necessary for the join to be based on disk.

Figure 4.3 shows the two datasets the scientists join. Axon cylinders represent 60% and dendrites 40% of the combined dataset of 250 million cylinders that model the neurons. As the illustration shows, the datasets differ significantly in data distribution: they have similar spatial extent but the axons are predominantly located at the top of the dataset. When joining theses datasets, areas of contrasting as well as similar density need to be joined efficiently, making an adaptive strategy key.

4.3 TRANSFORMERS Overview

As we demonstrate with the motivation experiment, each existing approach is efficient in joining a particular combination of dataset densities but none can join all combinations of data densities efficiently. The reason lies in their design: current approaches either use data-oriented partitioning (efficient for contrasting densities) or space-oriented partitioning (efficient for similar densities) but cannot take into account the variations of distributions and cannot adapt their join strategy.

We consequently design and use at the core of TRANSFORMERS *adaptive exploration* that robustly adapts the join strategy as well as the data layout at runtime. The join strategy is adapted by using the locally sparser data to guide the join, i.e., TRANSFORMERS uses the locally sparser data to selectively retrieve from the locally denser data only the elements needed, thereby ensuring that as little data as possible is retrieved and that as few elements as possible are tested for intersection. In case dataset A is locally sparser than dataset B,

TRANSFORMERS will use the area in A to guide the join. If the roles are switched, i.e., the area in B is locally sparser, it uses B as a guide. Additionally, if the contrast in density is substantial, TRANSFORMERS splits a locally sparse area in A into finer-grained units so that each unit only needs to be joined with a small, fine-grained subset of the area in the locally denser dataset. Adapting the join strategy to the local characteristics of both datasets ensures a robust performance.

More precisely, to perform a join given two indexed datasets, TRANSFORMERS randomly picks one dataset A and uses it as the *guide* while the other is used as the *follower*. The areas $a \in A$ of the guide are visited one after the other while the connectivity information in the follower is used to navigate through it and to move to the corresponding location in the follower. The area a used for navigation is called *pivot*. Once TRANSFORMERS arrives at the location of a pivot a, it uses crawling based on the connectivity information [97, 127] to detect all spatial elements of the follower that intersect with pivot a, and then continues exploration towards a neighboring area in the guide dataset.

TRANSFORMERS adapts its strategy at runtime by switching the guide and follower: if a very sparse area is joined with a dense one, it uses the sparse area as a *guide* and the dense dataset as *follower*. Switching the roles of the datasets at runtime ensures that we can always use the locally sparser dataset to retrieve as little data as possible from the locally denser dataset, thereby robustly curbing the amount of data read and the number of comparisons.

Crucially, TRANSFORMERS also adapts the data layout on the fly: if the areas compared from both datasets have a very different number of elements, it adapts the data structure and splits the sparse pivot area into finer-grained units and joins them individually with the dense follower, retrieving only exactly the data needed. If, on the other hand, the two datasets' density is locally similar, it groups spatial elements into larger groups and joins them as a batch, thereby curbing the overhead that would result from very fine-grained partitioning, i.e., repetitive reads and comparisons. Adapting the data layout reduces the data read (and unnecessary comparisons) and thereby levels fluctuations in the join time resulting in a more robust execution time of the join.

While TRANSFORMERS borrows elements from previous approaches, i.e., data-oriented partitioning from the R-Tree, crawling from GIPSY and joining big partitions from space-oriented partitioning approaches, the departure from the state of the art lies in its ability to adapt (a) its strategy, and (b) the data layout on the fly, thereby accomplishing robustness as well as substantially improved performance.

Figure 4.4 illustrates how TRANSFORMERS adapts strategy and data structures to the characteristics of the datasets while performing the join; it starts with one of the datasets and uses the connectivity information to move through the dataset. Once it arrives at the position of a pivot where the follower is sparser than the guide, it switches their roles, so it joins the locally sparse with the dense area (Transform 1 in Figure 4.4). When it detects areas of similar local density in guide and follower it uses a coarse-grained layout (Transform 2 in Figure 4.4).

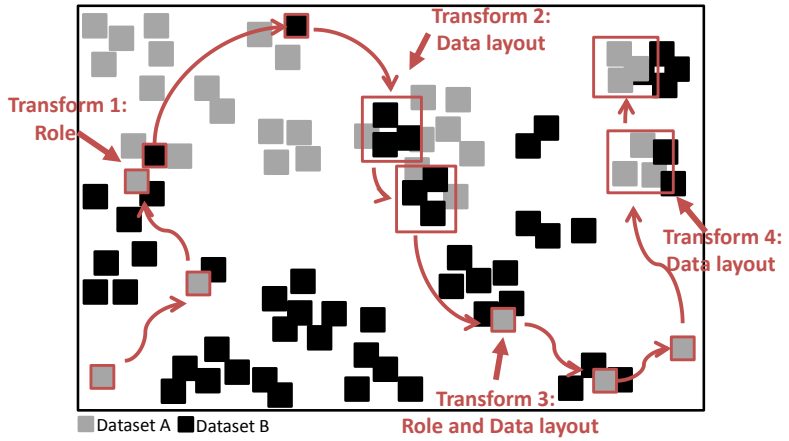


Figure 4.4 – TRANSFORMERS adapts to dataset characteristics.

Clearly, instead of adapting the join strategy at runtime, we could also adapt the partitioning of datasets A as well as B and use a static join strategy. The partitioning of dataset A, however, depends on the partitioning of dataset B (and vice versa) and so the adapted partitioning can only be used to join two specific, predetermined datasets. Adapting the join strategy, on the other hand, does not depend on a particular combination and therefore enables TRANSFORMERS to reuse partitioned datasets, amortizing the overhead over several joins.

4.4 TRANSFORMERS Indexing

To enable the *adaptive exploration* TRANSFORMERS requires (a) both datasets to be partitioned and (b) connectivity information between partitions. To overcome the issues of space-oriented partitioning, TRANSFORMERS first uses fine grained data-oriented partitioning on both datasets. To enable the adaptive exploration, it further computes connectivity information between partitions, i.e., it stores for each partition a list of adjacent partitions.

Partitioning. TRANSFORMERS uses a data-oriented partitioning approach similar to STR [70] to partition the datasets. It first sorts the dataset on the x-dimension of the element center and partitions the elements along this dimension. All resulting partitions are then sorted on the y-dimension and partitioned again. The resulting partitions are sorted on the z-axis, partitioned and each partition is stored on a disk page.

The aforementioned approach for data-oriented partitioning preserves spatial locality, i.e., spatially close elements are stored on the same disk page. Likewise, by choosing the size of the partitions at every step of the partitioning process, we can precisely determine the size of the final partitions. This (a) ensures that a partition can fit on a disk page (4K or a multiple thereof) and (b) gives us a parameter to control the granularity of the partitioning.

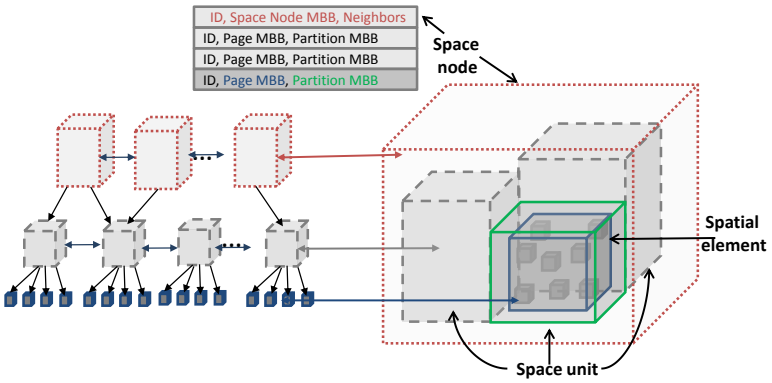


Figure 4.5 – The data structures: space node, space descriptor and space unit.

Data Organization. TRANSFORMERS produces two types of partitions; it first applies the partitioning algorithm on spatial elements producing *space units* and second, it groups the space units into *space nodes* using the same partitioning algorithm. The indexing phase thus produces a three-level hierarchical organization where level zero consists of space nodes, level one corresponds to the space units and level two to the individual spatial elements as illustrated in Figure 4.5 (left).

TRANSFORMERS stores the spatial elements on disk as space units: elements that belong to the same space unit are stored on the same disk page. It also stores meta information about each space unit in a *space descriptor*. A space descriptor summarizes a space unit *su*, i.e., it stores a pointer to the corresponding disk page, *su's* partition MBB and *su's* page MBB. The page MBB is the minimum bounding box containing all elements in a space unit (and thus on a disk page), whereas the partition MBB encloses the partition. Storing both, the page and the partition MBB, is necessary to ensure the correctness of the join process. Without the partition MBB there may be gaps between two neighboring pages MBBs (and thus the space units) in one dataset and TRANSFORMERS cannot navigate between them to explore and join the pages with the pages of the second dataset.

Finally, we group the neighboring space unit descriptors into *space nodes* that consequently store metadata information about the groups of partitions. A space node is also described with a *space descriptor*, i.e., the node's MBB that covers all its partitions and the neighbors of a space node. Figure 4.5 illustrates the data structures.

Connectivity. TRANSFORMERS computes the connectivity information by performing a spatial self-join on the space node MBBs, resulting in a list of all overlapping or adjacent nodes per space node. Any spatial join approach can be used for the self join. We use PBSM primarily because of its efficiency in the building phase. To decrease the amount of metadata necessary to be stored, a space unit inherits this neighborhood information from its parent space node.

Algorithm 4: Adaptive Walk Algorithm

Input: startFr: start descriptor in follower dataset
pivot: space node/unit/element
Output: clFr: closest space descriptor to pivot

$clFr = startFr$
enqueue $startFr$ into $fqueue$
while $fqueue \neq \emptyset$ **do**
  dequeue follower record fr from $fqueue$
  $dist = distance(fr.partitionMBB, pivot)$
  if $dist == intersection$ **then**
    return fr
  end
  if $dist < distance(clFr.partitionMBB, pivot)$ **then**
    $clFr = fr$
  end
  if $fqueue == \emptyset$ AND $!isMovingAway(clFr)$ **then**
    enqueue $clFr$'s neighbors that have not been checked in $fqueue$
  end
end
return $noIntersection$

4.5 TRANSFORMERS Join

Given two indexed datasets, TRANSFORMERS starts an adaptive exploration to detect intersecting pairs of spatial elements. It visits the elements of the locally sparser guide dataset, one after the other, navigating or *walking* between them using the connectivity information in the locally denser follower dataset. Once it arrives at the location of a particular guide element p, it *crawls* the neighborhood area to detect all elements of the follower that intersect with p using a *in-memory join*. Depending on the data layout used at this stage, an element p can represent either a space node, a space unit or a spatial element. TRANSFORMERS adjusts the data layout and the roles before the crawling step, to zoom into the area of interest.

Adaptive Walk. TRANSFORMERS first randomly assigns the roles of guide and follower to the datasets and then chooses a first pivot element, p, in the guide dataset. To determine what elements of the follower intersect with p, it needs to find a start space descriptor of the follower dataset as close as possible to p. It then uses the connectivity information to explore, i.e., recursively read all neighboring space descriptors and pick the one closest to p. Exploration is repeated until a space descriptor intersecting with p is found. If no neighbor descriptor closer to p can be found (the adaptive walk is moving away from p) and the partition MBB of the closest descriptors still does not intersect with p, then p does not intersect with any element of follower. The process is illustrated in Algorithm 4.

To initially find a start space descriptor as close as possible to p (to reduce the exploration overhead), we index the Hilbert value of the center point of all space nodes in a dataset with a B+-Tree. We use B+-Trees instead of an R-Tree (or similar indexes) to avoid the issue of overlap and also to speed up building the index. To find the descriptor, TRANSFORMERS formulates a range query based on the Hilbert values of the centers of two neighboring space nodes; it only uses the B+-Tree to find the starting point of the exploration. Alternatively, the first space node of the follower dataset can be used.

Adaptive Crawling. The crawl phase starts once an *intersection record* is found, i.e., a follower space descriptor whose partition MBB intersects with p. The goal of the crawl phase is to provide a *candidate set* for the final phase of the adaptive exploration process, that is, retrieving and testing actual spatial elements for intersection. Starting with the *intersection record*, similarly to the previous walk phase, the crawl phase recursively visits all neighbors until no more elements intersecting with p can be found. More precisely, it starts with the *intersection record* and recursively retrieves all linked neighbor records. If a space descriptor's *page MBB* intersects with p, then its space unit page is included in the *candidate set*. On the other hand, the neighbors of a space descriptor are visited, if and only if, not only its *page MBB*, but also its *partition MBB* intersects with p. The crawl phase thus ends when no more crawl records with a partition MBB intersecting with p can be found. Then TRANSFORMERS moves to the next element in the guide dataset.

In-memory Join. Once TRANSFORMERS processes an entire space node, it joins the detected follower's *candidate set* with the *pivots* that belong to the processed space node. It partitions space in a uniform grid and assigns the elements, belonging to the *pivots*, to the cells they overlap with. Finally, it probes the grid with the elements from the *candidate set* to find pairs of intersecting elements [129]. When TRANSFORMERS uses the node level as data layout it additionally filters elements before the in-memory join. It joins the page MBBs from the guide's and follower's *candidate set* to filter out space units that do not intersect with each other, thereby reducing the data read and compared.

The pseudocode of the adaptive exploration is shown in Algorithm 5. TRANSFORMERS is finished only once all the elements from one dataset are checked for intersection. The condition $isChecked$ varies depending on the current level, i.e., for the node level we check if one dataset is fully traversed and for the unit/object level if all the elements belonging to the enclosing node/unit are checked for intersection. If at the end of the initial pass both datasets have unexamined elements the adaptive exploration process restarts taking as a guide the dataset with fewer unexamined elements. The process continues until one dataset is fully traversed, guaranteeing that all intersections are found. TRANSFORMERS collects information about all elements in the dataset during the indexing phase (space node ids). It reuses this information as a to-do list during the join process to track checked elements. Depending on the current layout, we mark a space node as checked if the node itself is checked for intersection or all elements that constitute the node are checked.

Algorithm 5: Adaptive Exploration Algorithm

Input: level: current data layout
Output: intersections: result pairs
or candidateSet: input for the batch join
Data: p: space node/unit/element

```
while !isChecked() do
    p = loadCurrentPivot()
    intersect = adaptiveWalk(p, startRecord)
    if intersect == noIntersection then
        continue
    end
    switch applyTransformation(p) do
        case NoTransformation do
            adaptiveCrawling(intersect, candidateSet)
        case RoleTransformation do
            switchGuideFollower() & continue
        case LayoutTransformation do
            switchLayout()
            adaptiveExploration(level++)
    end
    if level == Node then
        join(p, candidateSet, intersections)
        removeFromToDoList(p)
    end
end
return intersections/candidateSet
```

Figure 4.6 illustrates how TRANSFORMERS uses the *elements* of the *guide* dataset to direct walking in the *follower* (for simplicity only space nodes and units are used). In this example, both datasets are grouped into partitions of three elements. TRANSFORMERS initially uses a coarse-grained layout (space nodes) and randomly chooses guide and follower dataset, e.g., A and B respectively. In *adaptiveWalk* it immediately detects an intersection between $a1$ and $b1$ and thus, before checking the actual elements for intersection and performing unnecessary reads and comparisons, it checks if it is necessary to *applyTransformation*. Considering that area $b1$ is significantly sparser compared to the same area in dataset A, TRANSFORMERS switches roles and adjusts the data layout: dataset B becomes the guide and space node $b1$ is split into space units, filtering out six partitions from dataset A. Once *adaptiveCrawling* and *join* are done, TRANSFORMERS resets the data layout to space node, keeps the dataset B as guide and uses $b2$ as next pivot, leading to additional transformations.

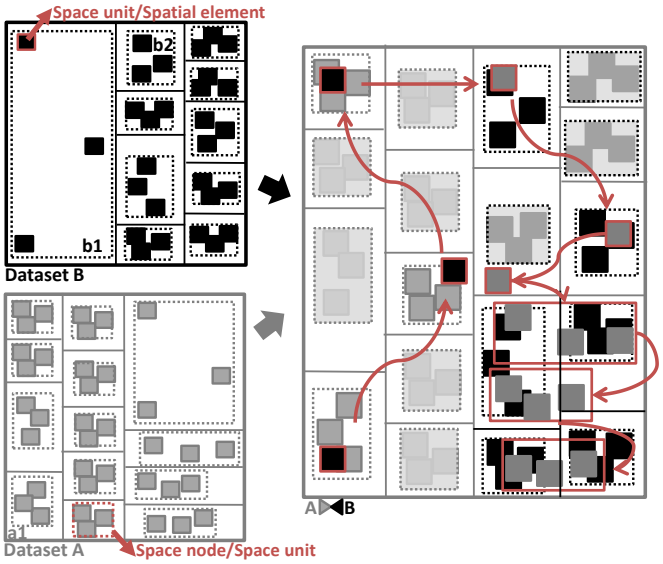


Figure 4.6 – Joining datasets A and B using adaptive exploration.

4.6 Transformations

Crucially, when TRANSFORMERS moves to a new pivot p in the guide dataset, it adapts its strategy by adjusting the roles of guide and follower and adapts the layout.

4.6.1 Role Transformation

When joining two datasets with skewed distribution the join approach needs to adapt to the data. The roles of *pivot*, *guide* and *follower* define the configuration of TRANSFORMERS; a proper combination of roles accelerates the join and makes its performance more robust compared to static approaches.

TRANSFORMERS thus adapts the roles of *pivot*, *guide* and *follower* at runtime based on the two datasets' density ratio. Considering that both datasets rely on the same indexing strategy and thus have the same number of elements in the corresponding space units/nodes, a significant difference in volume of space units/nodes indicates that one area is sparser than the other. TRANSFORMERS thus uses the volumes enclosed by the *elements* (space node/unit) of the guide V_g and follower V_f datasets, at the location of the pivot in both datasets, to compute the ratio V_g/V_f. If the ratio is smaller than a threshold t it first switches the roles, i.e., the guide becomes the follower and the follower the guide, and then also changes the pivot (picks the element in the new guide closest to the old pivot). By adapting the roles, TRANSFORMERS ensures that it can always use the sparser dataset as guide and thus only retrieves the data needed from the denser dataset. This decision is followed by data layout transformation.

4.6.2 Data Layout Transformation

TRANSFORMERS also adapts the data layout at runtime to further reduce data read and comparisons performed. It initially designates a space node as the first pivot, i.e., it uses a coarse-grained data layout in both datasets. During the join process it may split the space node into finer-grained data structures, as a coarse-grained data layout is not a good strategy for adaptive exploration when joining areas of divergent densities. That is, in case the pivot is sparser than the corresponding area in the follower, it moves the pivot down to a level of granularity which allows for better filtering in both datasets. At runtime, TRANSFORMERS tests the ratio V_g/V_f between the datasets and changes the data layout if it is above threshold t. This decision is potentially preceded by a role switch between a guide and follower dataset if we detect a locally sparser area in the follower.

TRANSFORMERS moves seamlessly between three data layouts, predefined/produced during the indexing phase, that correspond to different levels of hierarchy as shown in Figure 4.7. It initially uses a coarse-grained data layout in both datasets and therefore performs adaptive exploration on level 0. In case of local density difference, it moves to a finer granularity by splitting space nodes into space units and thus performs adaptive exploration on level 1, in both datasets. Furthermore, if it detects substantial local density difference on a space unit level, it switches to the finest-grained data layout: it splits a space unit into its spatial elements, thus using a spatial element as pivot (level 2) while using the space unit as a level of granularity for the follower (level 1). TRANSFORMERS does not split the follower to object granularity as keeping track of connectivity information at this level causes significant overhead. Crucial for the data layout transformation is to transform pivot to finer granularity. We also transform the follower, when possible, to allow for better data filtering and skew detection. TRANSFORMERS decides what the data structure element e is, i.e., space node, space unit or element, based on the pivot. The same data structure is used until the end of the exploration phase, i.e., until a new pivot is chosen.

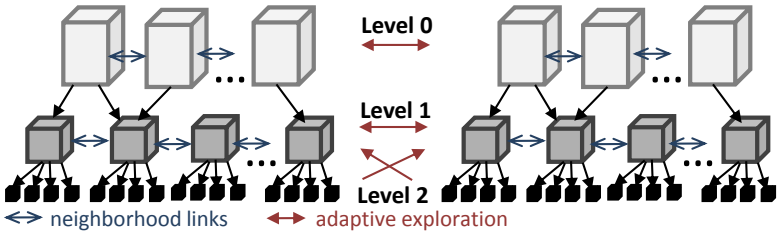


Figure 4.7 – The hierarchical organization of TRANSFORMERS.

The number of levels TRANSFORMERS uses and their granularity is primarily driven by its design for use on disk. To optimize access on disk, we align data structures for disk and ensure they are page-aligned. Two levels are given: Level 2 is given by the single elements which cannot be further split as they are the smallest spatial primitive used. A second level is defined by the actual storage structure on disk: to optimize access to disk, we pack as many elements

into a space unit as can fit on a disk page giving us level 1. Finally, we add a third level (level 0) that summarizes several space units on level 1 into space nodes. Level 0 is again designed page aligned, i.e., as many level 1 space units as can be summarized and stored on a disk page are combined into level 0 nodes.

TRANSFORMERS can introduce more levels between the existing ones or use more levels to recursively summarize level 0 (e.g., level -1 etc.). The former, introducing levels between existing ones, however, is inefficient as the resulting data structures would no longer be page-aligned, therefore retrieving unnecessary data (or half empty pages). Recursively summarizing level 0 leads to space nodes on higher levels which have a considerable spatial extent that soon makes the spatial extent of higher levels nodes indistinguishable from space-oriented partitioning (thereby also inheriting the same performance issues).

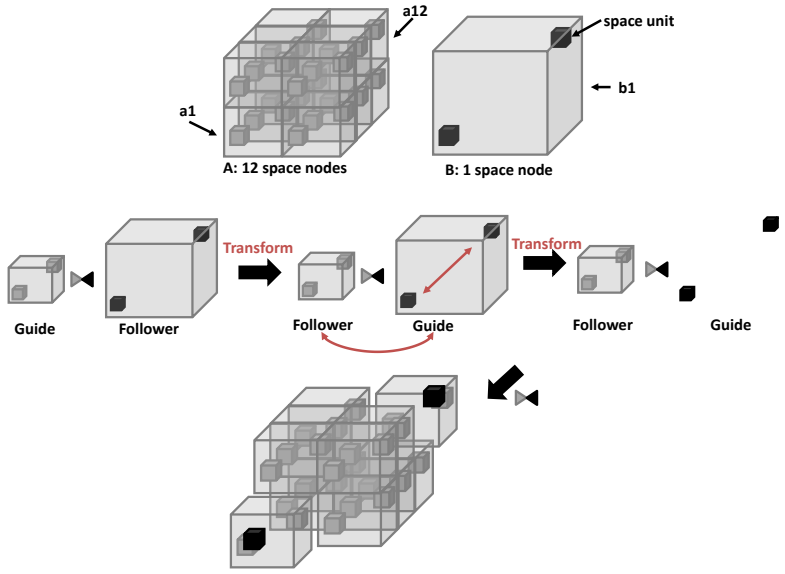


Figure 4.8 – Role and Data layout transformation.

Figure 4.8 illustrates the result of the partitioning where all space nodes contain two space units. For the sake of simplicity we do not illustrate spatial elements. Dataset *A* represents a densely populated area that fits 12 space nodes in the same space while *B* is sparsely populated containing one space node. According to the default adaptive exploration strategy, the space node *a1* will be used as a pivot to check the corresponding area (space node *b1*) of the dataset *B*. Without data layout transformation, the next pivot would be *a2* and eventually we would load and test all space units/elements in *a1*-*a12* against *b1*. This is unnecessary considering that *b1* overlaps only with *a1* and *a12*. A better exploration strategy in this case is to execute the exploration directed by the space units in the sparse area. TRANSFORMERS therefore switches the roles and splits the pivot, moving to a finer granularity. It then uses dataset *B* as a guide and a spatial unit as pivot. By doing so it decreases the number of disk accesses and comparisons of spatial elements. At the same time, the overhead in the number of metadata

comparisons is increased. Metadata comparisons, however, are not expensive as we will show in the experiments.

4.6.3 Transformation Thresholds

Setting the thresholds for transformations is important for TRANSFORMERS' performance. In particular, we need to determine the thresholds for changing the data layout, from space node to space unit and from space units to single spatial elements, and the threshold for the role transformation.

Data Transformations Threshold. The first threshold we need to determine is when TRANSFORMERS has to switch from a coarse-grained granularity (space node) to a finer-grained granularity (space unit). As described in Section 4.6.1, we compare the corresponding volumes in the guide and follower datasets and if the ratio exceeds a threshold t_{su} ($V_g/V_f \geq t_{su}$) we split (transform data layouts). To determine the threshold t_{su}, we first define the cost and benefit of the splitting operation.

Equation 4.1 defines the additional cost as the adaptive exploration (splitting means more elements to traverse), i.e., the number of new space units (nSU), after splitting a space node, times the cost of traversal and exploration (T_{ae}).

$$nSU \times T_{ae} \tag{4.1}$$

The average benefit of splitting, on the other hand, is essentially time saved by reading fewer space units ($nSU \times T_{io}$) and testing fewer spatial elements for intersection ($nSU \times nSO \times T_{comp}$, where nSO is the number of spatial elements in a space unit). How many space units do not need to be considered is difficult to define a priori. The ratio V_g/V_f corresponds to the maximum number of space units that can be filtered out. We adjust this value using the parameter $c_{flt} = (0,1)$, determined at runtime based on the actual percentage of filtered elements. Equation 4.2 formalizes the benefit of splitting.

$$\frac{V_g}{V_f} \times c_{flt} \times nSU \times (T_{io} + nSO \times T_{comp}) \tag{4.2}$$

Clearly, if the benefit exceeds the cost, then we should split the space (Equation 4.3). Equation 4.4 therefore defines the corresponding threshold t_{su}. T_{ae}, T_{io} and T_{comp} are all parameters that heavily depend on the hardware of the system and are therefore best determined at runtime. TRANSFORMERS initially uses the default threshold values (Section 4.7.7) that are updated after the first transformation.

$$\frac{V_g}{V_f} \geq \frac{T_{ae}}{c_{flt} \times (T_{io} + nSO \times T_{comp})}; \frac{V_g}{V_f} \geq t_{su} \tag{4.3}$$

$$t_{su} = \frac{T_{ae}}{c_{flt} \times (T_{io} + nSO \times T_{comp})} \tag{4.4}$$

Role Transformations Threshold. The data layout transformation is potentially preceded by a role switch between the guide and the follower dataset if we detect that a locally sparser area belongs to the follower dataset, that is:

$$\frac{V_f}{V_g} \geq t_{su}; \frac{V_g}{V_f} \leq \frac{1}{t_{su}}; \frac{V_g}{V_f} \leq t_{suRole}; t_{suRole} = \frac{1}{t_{su}} \tag{4.5}$$

Finest-grained Data Transformations Threshold. Once we are on a space unit level we can additionally adapt the data layout if we detect "extreme skew", i.e., a considerable V_g/V_f ratio. Similarly to deciding whether to split a space node into a space unit, we need to decide if we split a space unit into single spatial elements. The reasoning behind cost and benefit is the same, except that we need to adjust the cost of adaptive exploration by using nSO (the number of spatial elements in a space unit) instead of nSU, as illustrated in Equation 4.6 and Equation 4.7.

$$nSO \times T_{ae} \tag{4.6}$$

$$\frac{V_g}{V_f} \times c_{flt} \times nSU \times (T_{io} + nSO \times T_{comp}) \tag{4.7}$$

The threshold t_{so} for deciding on splitting further thus is the ratio between the new cost and new benefit (Equation 4.8).

$$t_{so} = \frac{nSO \times T_{ae}}{nSU \times c_{flt} \times (T_{io} + nSO \times T_{comp})} \tag{4.8}$$

4.7 Experimental Evaluation

In this section we describe the experimental setup & methodology, compare TRANSFORMERS against state-of-the-art spatial join approaches and then analyze its performance. To study the impact of different dataset characteristics on the performance of TRANSFORMERS we use synthetic datasets where we control the number, size and distribution of the elements. As a final test we use neuroscience datasets to compare performance on a real workload.

4.7.1 Experimental Setup

Hardware. We run the experiments on Red Hat 6.3 machines equipped with 2 quad CPUs AMD Opteron, 64-bit @ 2700 MHz, 32 GB RAM and 4 SAS disks of 300GB (10000 RPM) capacity as storage. We use one of the disks for the experiments, i.e., no RAID configuration is used.

Software. All algorithms are implemented single-threaded in C++ for a fair comparison.

Setting. We experimentally compare TRANSFORMERS against the latest or most broadly used spatial joins, i.e., the Partition Based Spatial Merged Join (PBSM), the Synchronized R-Tree Traversal (R-TREE), and GIPSY. Like the approaches we compare it with and driven by our

motivating application, TRANSFORMERS is designed to join two static spatial datasets and we do not compare it to self-joins or trajectory joins. PBSM and TRANSFORMERS use the grid hash join [129] as the in-memory join algorithm, while R-TREE uses the plane sweep. R-TREE is based on R-Trees bulkloaded using the STR approach [70]. While more sophisticated approaches can outperform STR for certain dataset characteristics (e.g., TGS [34] and PR-Tree [10]), they incur considerable overhead for partitioning the data. In practice STR balances the overhead of partitioning the data and the size of MBBs (and thus the overlap) well.

Given the absence of heuristics, we set the configuration of all approaches other than TRANSFORMERS for the best performance identified with a parameter sweep. For PBSM the configurations with 10^3 (uniform and clustered distribution) and 20^3 (neuroscience data) partitions balances the number of elements needed to be compared by the grid hash join algorithm and the number of elements replicated, deduplicated, additionally written/read to/from disk best, and therefore executes the fastest. The synchronized R-Tree approach (R-TREE) uses a fanout of 135 (based on disk page size). The parameters of TRANSFORMERS are set according to Section 4.6.3.

We set the disk page size to 8KB for all approaches. For all experiments we assume cold system caches and we therefore clear OS caches and disk buffers before each experiment.

4.7.2 Experimental Methodology

Synthetic Datasets. We create synthetic datasets by distributing spatial boxes in a space of 1000 units in each dimension of a three-dimensional space. The length of each side of each box is determined uniform randomly between 0 and 1. The spatial elements are distributed using a particular data distribution. We use two basic data distributions - clustered and uniform.

We use three different types of clustered datasets which differ in number and size of the clusters. For the *DenseCluster* we produce on average 700 densely populated clusters. *UniformCluster* datasets contain 100 clusters whose elements are distributed in a wide area resulting in a nearly uniform distribution while *MassiveCluster* datasets contain 5 densely populated clusters each with a fixed number (100K) of uniformly distributed elements. *DenseCluster* and *UniformCluster* use a normal distribution ($\mu = 500$, $\sigma = 220$) to determine the centers of the clusters. Figure 4.9 illustrates the datasets.

The number of spatial elements in the datasets ranges between 100M and 1300M (50M-650M per dataset) resulting in a size on disk between 4.6GB and 58.2GB.

Neuroscience Datasets. To evaluate TRANSFORMERS on real data we use a small part of the rat brain model represented with 450 million cylinders as elements. We take from this model a contiguous subset with a volume of 285 μm^3 and approximate the cylinders with minimum bounding boxes. In the spatial join process axons are represented by one dataset, dendrites by another and the detected intersections represent the synapses. The number of spatial

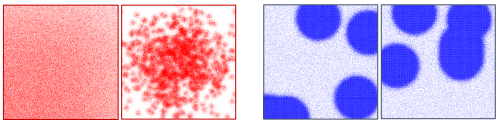


Figure 4.9 – *UniformCluster* & *DenseCluster* (left) and *MassiveCluster* (right) dataset samples.

elements joined ranges between 100M to 500M (50M-250M per dataset). The size on disk ranges from 5GB to 16GB.

Approach. Spatial joins typically involve two steps: filtering followed by refinement. The filtering step finds pairs of spatial elements whose approximations (MBBs) intersect with each other, while the refinement step detects the intersection between the actual shape of the elements. Considering these two steps are independent in terms of their implementation and the refinement step is application specific, we focus on the filtering like most spatial join methods [58] and like other evaluations we do not account for the refinement step.

We first perform comparative analysis, i.e., we evaluate the performance of TRANSFORMERS and compare it with other approaches (PBSM, R-TREE, GIPSY) in four sets of experiments. More precisely, we first expand the motivation experiment with TRANSFORMERS demonstrating its robustness on uniform datasets. We then evaluate the performance of spatial joins on datasets with non-uniform data distributions, on uniform distributions and finally demonstrate TRANSFORMERS benefits on real neuroscience data. For the latter experiments we measure the time to index, a breakdown of the join time and the major factor of the join time, the number of intersection tests between spatial elements. We also perform the sensitivity analyses of TRANSFORMERS, i,e., we analyse the impact of transformations and quantify the overhead of adaptive exploration.

4.7.3 Robustness

This set of experiments illustrates TRANSFORMERS' robustness with respect to varying relative density. Figure 4.10 illustrates the performance of TRANSFORMERS when performing the set of experiments from Section 4.2.1. The results of the join, excluding the index building time, are shown in Figure 4.10. The values above the curves indicate the density ratio of the datasets. TRANSFORMERS outperforms GIPSY when joining datasets with the highest density ratio (point 1000x) with a speedup of 5, while its speedup over PBSM is 6.7 when joining two dense datasets (point 1x). Its average speedup over the R-TREE is 10.

TRANSFORMERS combines different levels of granularity and switches to the finest level only when the walking overhead is low. GIPSY's performance, on the other hand, suffers from the overhead of the directed walk on the spatial element level, which is its only level of granularity. By being able to combine coarse (space node) and fine-grained granularity (space unit), TRANSFORMERS compares less data than PBSM. The performance of PBSM

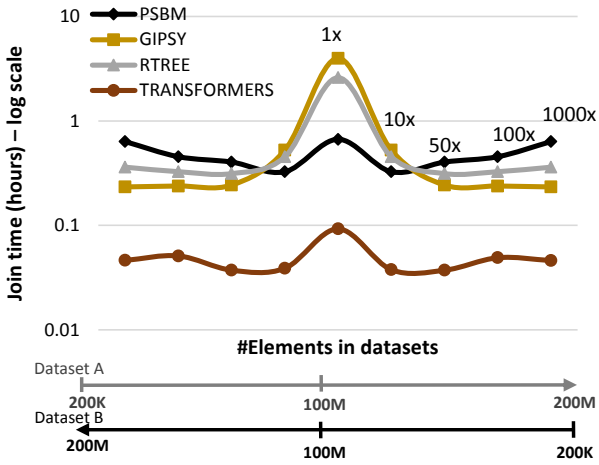


Figure 4.10 – Joining datasets with variable relative density.

is also significantly affected by random reads: PBSM writes pages to disk arbitrarily while indexing (when the number of elements buffered for a cell exceeds the disk page size) leading to random reads when retrieving elements in a cell.

This experiment exemplary demonstrates the robustness of TRANSFORMERS: by adapting to dataset characteristics at runtime (changing role and data layout), the join algorithm can compensate for extremely contrasting dataset densities.

4.7.4 Non-uniform Data Distributions

In the following set of experiments we compare and analyse TRANSFORMERS' performance on datasets with non-uniform distribution. We join synthetic datasets with clustered distribution: one dataset corresponding to *DenseCluster* and one to *UniformCluster*. We increase the size of the datasets from 350M to 650M elements, in steps of 100M and measure join and indexing time. Due to the long execution time when joining densely populated datasets, we exclude GIPSY from all these experiments and R-TREE when joining the biggest datasets (650M elements).

Indexing. The index building time is the time necessary to build the initial data structures. For PBSM this process involves creating partitions and assigning elements to them and for TRANSFORMERS partitioning the data, organizing metadata information and introducing connectivity information. Similarly to TRANSFORMERS, R-TREE has to partition the space and additionally build all levels of the hierarchy.

Figure 4.11 shows the results of measuring the indexing time. The results illustrate the difference between indexes based on space- and data-oriented partitioning very well. As a space-oriented approach, PBSM only needs to assign each element to the cells of a uniform

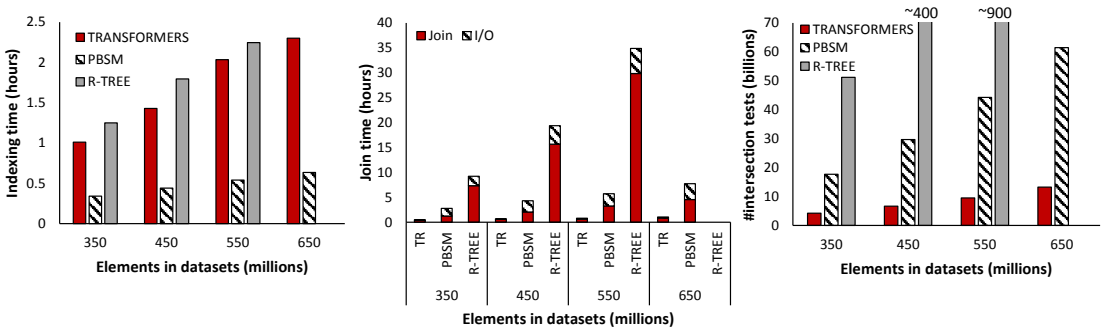


Figure 4.11 – Execution time breakdown and number of intersection tests for the join phase on synthetic data.

grid they overlap with. PBSM consequently outperforms TRANSFORMERS by a factor of between 2.9 - 3.6. As a data-oriented partitioning approach, on the other hand, TRANSFORMERS spends time on creating data partitions of equal size. It essentially needs to sort the spatial elements in three dimensions to produce partitions (that correspond to the space units). R-TREE partitions the data with a similar strategy but additionally has to recursively build levels, resulting in a higher indexing time.

While PBSM efficiently partitions the data, the partitions produced are unlikely to be reused efficiently. The resolution of the grid of PBSM is determined based on several factors (number of elements, spatial extent and distribution of elements) but crucially on the size of the elements of both datasets because the size of the grid cells needs to be chosen so that not too many elements are replicated. The partitions produced therefore depend on the characteristics of a particular combination of datasets and cannot efficiently be reused when joining with datasets that have considerably different characteristics.

As opposed to PBSM, TRANSFORMERS builds the indexes for each dataset separately and adapts the join execution to the characteristics of the two indexes. An index built on one dataset can therefore be reused when joining with any other dataset. The additional time TRANSFORMERS requires to index one dataset can therefore be amortized over additional joins with other datasets.

Join Performance. To analyse the join performance we break the execution time into I/O and join time. The I/O time is the time spent on loading data during the join process while join time is the time needed to join the data in memory, i.e., testing spatial elements for intersection (and related operations). The results of the experiments joining two datasets of the same size are shown in Figure 4.11 (middle). TRANSFORMERS (labeled TR) achieves the best results and outperforms PBSM by a factor between 5.5 & 7.4.

PBSM requires substantially more time for I/O because it reads a significant amount of unnecessary data during the join phase. Data-oriented partitioning in combination with adaptive exploration allows TRANSFORMERS to filter out 20% of the total data on average when joining

	TRANSFORMERS	PBSM	RTREE
150M	0,16	1,02	4,55
250M	0,30	2,24	11,63
350M	0,49	4,28	24,92

Table 4.1 – Execution time (hours) for datasets with uniform distribution.

DenseCluster datasets. When comparing the datasets with more distinctive local variations (e.g., *MassiveCluster*) TRANSFORMERS filters out on average 47% of the data while PBSM has to read all data. The execution time, however, is additionally determined by the join selectivity and the randomness of reads. PBSM inherently writes data for each partition to disk individually (and thus distributed) in the indexing phase, resulting in almost exclusively random reads during the join phase.

Figure 4.11 (right) shows the number of intersection tests between spatial elements for the same experiment. For TRANSFORMERS this time also includes metadata comparisons. PBSM compares all elements in one (possibly large) cell of both datasets and thus only avoids tests between elements in different cells. PBSM consequently needs to perform 4.4 times more comparisons than TRANSFORMERS which only retrieves and compares the very fine-grained partitioned data.

Given the very efficient indexing of PBSM, the overall improvement of TRANSFORMERS when taking into account the indexing and the join phase over PBSM shrinks to 2.1 - 2.5. More important, however, is the speedup in the join phase as the indexes built for TRANSFORMERS can be reused for future joins (between different datasets).

4.7.5 Uniform Data Distributions

To further demonstrate TRANSFORMERS general applicability, we join *Uniform* datasets with a uniform distribution (similar distribution and density throughout the area they cover). We vary the number of elements from 150M to 350M, in steps of 100M. The results of the join are shown in Table 4.1.

For this set of experiments TRANSFORMERS achieves a improvement of between 6.2 - 8.6 compared to PBSM. The overall improvement when joining datasets with a uniform distribution is a result of TRANSFORMERS' initial strategy that suits the datasets with similar distribution and similar number of elements. In addition, its preprocessing strategy provides sequential access to data as it preserves spatial proximity. Considering that we join densely populated datasets with the uniform distribution, PBSM's performance deteriorates compared to the previous set of experiments due to the increased replication rate. By default, its strategy causes elements replication that leads to additional I/Os, comparisons and deduplication. Although the grid hash join provides better performance for PBSM than the plane sweep join, it additionally

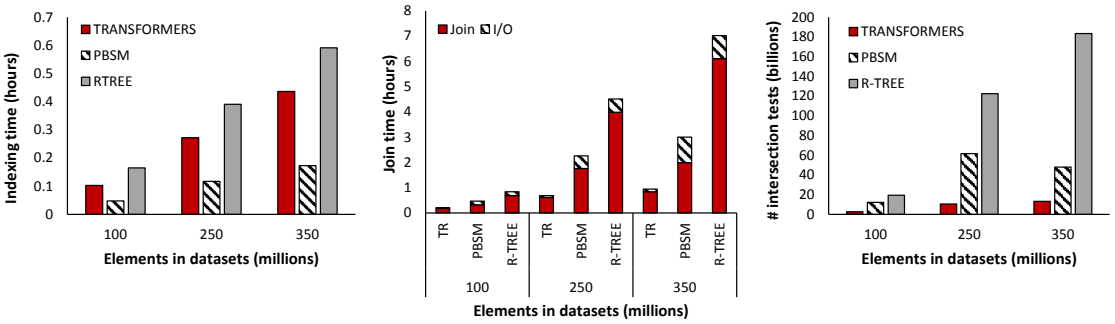


Figure 4.12 – Execution time breakdown and number of intersection tests for the join phase on neuroscience data.

increases the replication rate. The R-TREE join suffers from overlap at tree level and therefore performs on average 21 times more comparisons.

4.7.6 Neuroscience Data

To demonstrate the general applicability of TRANSFORMERS we also test its performance on neuroscience data by performing joins like the neuroscientists do. In total, at most 350 spatial elements are spatially joined, where 250M are axons and 100M are dendrites.

As the illustration shows (Figure 4.3) the neuroscience dataset has a skewed distribution and hence TRANSFORMERS behaves similarly as in the previous set of experiments. Figure 4.12 illustrates the experimental results. TRANSFORMERS achieves a speedup in join time of 2.3 - 3.3 compared to PBSM and 4.1 - 6.5 compared to R-TREE.

4.7.7 TRANSFORMERS Analysis

In the following we analyse the impact of transformations and quantify the overhead of adaptive exploration.

Impact of Transformations. In the following experiments we use *MassiveCluster* datasets to illustrate the impact of transformations on the performance. We join the same datasets, once with TRANSFORMERS and once with TRANSFORMERS that does not apply transformations (No TR), i.e., it just uses space nodes as the level of granularity.

With the increase of dataset size, also the data skew increases for *MassiveCluster* datasets. As the results measuring the join time in Figure 4.13 show, the benefit of transformations increases as the skew grows. An increase in skew triggers finer-grained transformations and thus TRANSFORMERS filters on average 47% of data resulting in an improvement in the performance between 1.2 and 1.6 compared to No TR.

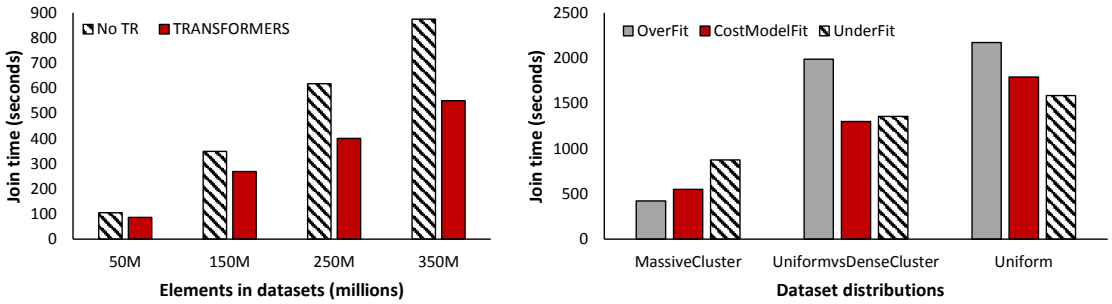


Figure 4.13 – Impact of transformations on join performance (left) and transformations threshold sensitivity (right).

Transformation Threshold. As discussed, TRANSFORMERS' performance depends on the transformation threshold. If the threshold is too high we will not benefit from transformations. On the other hand, if we set the transformations threshold too low the performance will be affected by the adaptive exploration overhead.

In these experiments we test the cost model using three datasets, each with 350M elements but with different data distributions: *MassiveCluster*, *UniformCluster* & *DenseCluster* and *Uniform*. To demonstrate the quality of the cost model, we use two additional configurations: *OverFit* uses 1.5 as a threshold and thus triggers many transformations and *UnderFit* uses 1,000,000 which prevents transformations, i.e., the algorithm uses default guide and follower and space nodes as data layout. All the parameters of the cost model are set and updated at runtime with an additional constraint that T_{ae} and c_{flt} are provided once the first transformation is executed. To trigger the first transformation we set the corresponding thresholds to initial values, i.e., $t_{su} = 8$, and $t_{su} = 27$. This volume ratio corresponds to the case where an edge of one MBB is two/three times bigger than the other one.

The results of the join are shown in Figure 4.13. When joining datasets with uniform distribution TRANSFORMERS should perform a minimal number of transformations since the two datasets do not have local variations in the distribution. The threshold proposed by the cost model leads to performance close to *UnderFit*, the best configuration tested. *MassiveCluster* datasets have significant local variations in the distribution and therefore benefit considerably from transformations. Therefore, the threshold proposed by the cost model provides a performance very close to *OverFit* (leading to many transformations). The data distribution in *UniformCluster* & *DenseCluster* datasets (empty areas in *DenseCluster*, Figure 4.9) allows the coarse grained configuration to filter considerable number of elements and the performance of the cost model and *UnderFit* is thus similar.

Adaptive Exploration Overhead. The adaptive exploration process potentially introduces overhead. More precisely, by using a fine-grained data granularity we may lose the benefit of processing comparisons in a batch operation and we may thus unnecessarily repeat filtering operations such as distance and overlap calculations.

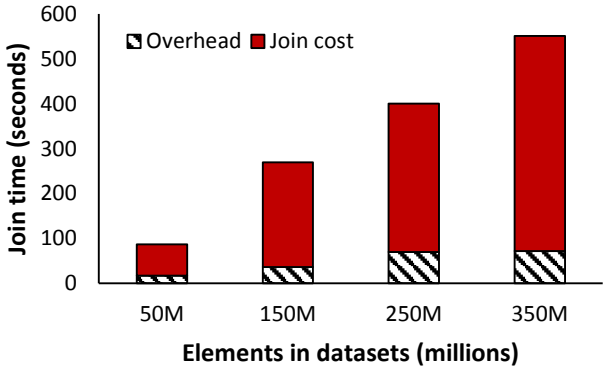


Figure 4.14 – Adaptive exploration overhead.

In the following experiments we measure the overhead of adaptive exploration using *MassiveCluster* datasets. To analyze the join performance, we break the execution time in join cost and adaptive exploration overhead. The join cost is the time spent on disk access and the time needed to join the data (the final *candidate set*) in memory. Everything else is considered as the overhead of adaptive exploration (Overhead).

As the results of the experiment show in Figure 4.14, the data layout transformations manage to keep the adaptive exploration overhead low by moving to a coarser granularity if too many elements need to be visited. On average, the adaptive exploration overhead takes 17% of the join execution time.

4.8 Conclusions

This chapter identifies the problem of spatial join *robustness*, which arises when joining spatial datasets of similar volumes but locally varying densities. As we show, current methods cannot efficiently perform a join between such datasets, which are prevalent in applications across sciences; such methods read too much data and/or require too many comparisons.

We propose TRANSFORMERS, a method that achieves robust spatial joins by adapting to local data characteristics. TRANSFORMERS partitions the data in advance, but, contrary to previous work, does not rely solely on that partitioning; it also adapts its execution in an on the fly, data-driven manner. First, it uses the locally sparser dataset to guide data retrieval, ensuring that only strictly needed data from the locally denser dataset are retrieved. Second, it adjusts the employed data layout, ensuring that only the relevant parts of the locally denser dataset are compared.

We show that TRANSFORMERS achieves robust and efficient joins. Thanks to its adaptivity, it achieves a speedup between 2 and 8 in the join phase compared to PBSM, the fastest state-of-the-art method throughout the density ratio spectrum. Moreover, it is scalable, capable to perform on ever bigger data of growing density variations.

Part II

Workload-Aware Spatial Incremental Indexing

5 Disk-Based Incremental Indexing

Advances in data acquisition – through more powerful supercomputers for simulation or sensors with better resolution – help scientists tremendously to understand natural phenomena. At the same time, however, it leaves them with a plethora of data and the challenge of analysing it. Ingesting all the data in a database or indexing it for an efficient analysis is unlikely to pay off because scientists rarely need to analyse all data. Not knowing a priori what parts of the datasets need to be analysed makes the problem challenging.

Tools and methods to analyse only subsets of this data are rather rare. In this chapter we therefore present Space Odyssey[1], a novel approach enabling scientists to efficiently explore multiple spatial datasets of massive size. Without any prior information, Space Odyssey incrementally indexes the datasets and optimizes the access to datasets frequently queried together. As our experiments show, through incrementally indexing and changing the data layout on disk, Space Odyssey accelerates exploratory analysis of spatial data by significantly reducing data-to-insight time compared to the state of the art.

5.1 Introduction

In astronomy, biology, neuroscience and other disciplines, scientists are increasingly overwhelmed by the amount of data they have at their disposal. With advances in sensor technology, i.e., increased resolution, and supercomputing for large-scale simulations, the amounts of data scientists have to analyse grow rapidly. Today's tools are frequently inadequate to analyse the data and answer key questions as already executing simple queries such as spatial range queries becomes challenging given the amount of data. Datasets can, of course, be indexed a priori to accelerate access but the areas analysed are rarely known beforehand and also only touch a subset of the entire dataset, making indexing an undue overhead.

In neuroscience, for example, scientists need to explore multiple massive datasets originating from different sources [82] to investigate particular areas of the human brain. The data in this

[1] Space Odyssey originally appeared in [101].

use case is spatial and originates from different instruments (e.g., patch clamp, brightfield spectroscopy, MRI) of different resolutions. To perform an analysis, they need to query small parts of different combinations of datasets, each of a size in the order of Terabytes. What areas of the datasets they need to access and what combinations of them are not known a priori. It is consequently unclear what parts of what datasets need to be indexed. It is however clear that fully indexing all datasets introduces considerable overhead which is unlikely to pay off.

More formally, the problem is the efficient exploratory analysis of multiple spatial datasets through the execution of range queries: given n datasets and a subset of datasets $m \subseteq n$, scientists need to efficiently execute a spatial range query q on each of the datasets m. What combinations of datasets m will be queried together and what spatial ranges q will be accessed is not known beforehand. The challenges thus are twofold (a) what areas in the datasets are accessed and (b) what datasets are accessed together.

Multiple spatial indexes have been developed to accelerate access to spatial datasets addressing the first challenge [33]. All of them, however, require the whole dataset to be indexed at once. Incremental approaches to indexing (or reorganising data layout) have been developed for one-dimensional data stored in main memory [53, 54], but not for spatial data on disk. The first challenge of incrementally indexing spatial data on disk as well as the second, accelerating access to multiple datasets queried together, remain unaddressed in literature to the best of our knowledge.

Space Odyssey, the approach we develop, addresses both challenges and enables the efficient exploration of multiple spatial datasets. While the datasets are being queried, it incrementally indexes datasets (based on space-oriented indexing) to accelerate access to the datasets in general and to the areas frequently queried in particular. At the same time, it reorganises the layout of the data on disk so that parts of the datasets queried together can be retrieved more efficiently. By incrementally indexing and reorganising the data, Space Odyssey accelerates explorative analysis of spatial data by substantially reducing data-to-insight time: Space Odyssey answers up to several hundred queries (more than half the queries of the benchmark) by the time the fastest existing approach has merely indexed the data.

The remainder of this chapter is organized as follows. We give an overview of our approach in Section 5.2. We then discuss in detail the incremental indexing strategy in Section 5.3 and the process of combining datasets in Section 5.4. We then analyse our approach experimentally in Section 5.5 before we conclude in Section 5.6.

5.2 Space Odyssey Overview

Space Odyssey enables exploratory access to multiple spatial datasets such that scientists can efficiently access particular areas in combinations of datasets. Crucially, our approach enables efficient access without having to preprocess the data. Instead, Space Odyssey uses incoming queries to reorganize the physical layout of the data to better serve queries.

First, to enable efficient access to precisely the areas queried in individual datasets, Space Odyssey incrementally indexes the datasets. Second, to better support querying the same areas in different datasets, it adapts the physical layout on disk, storing together the areas that are queried together to accelerate retrieval.

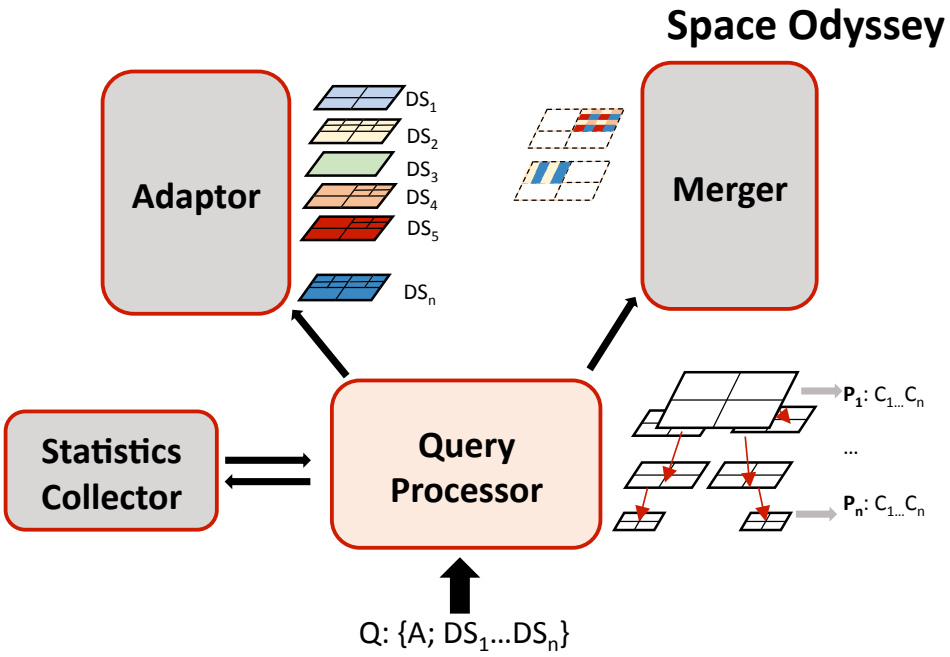


Figure 5.1 – Space Odyssey: components, data structures and a snapshot of the physical layout.

Figure 5.1 illustrates the architecture of Space Odyssey – its components, data structures and a snapshot of the physical layout. The Adaptor is responsible for the incremental indexing and the Merger performs operations related to the physical layout. Finally, the Query Processor orchestrates the overall query execution using information provided by the Statistics Collector.

5.3 Incremental Indexing

Indexing all datasets a priori has the major drawbacks that (a) scientists must wait until all data is indexed before they can start to query and (b) data that is never queried is indexed in a time-consuming process.

Space Odyssey therefore uses incremental indexing where in every step (with every query) we additionally refine the index structure in the frequently queried "hot" areas to accelerate future queries. At the same time, to keep the overhead of incremental indexing low, we use space-oriented indexing, as it introduces minimal processing overhead (compared to data-oriented partitioning [33]).

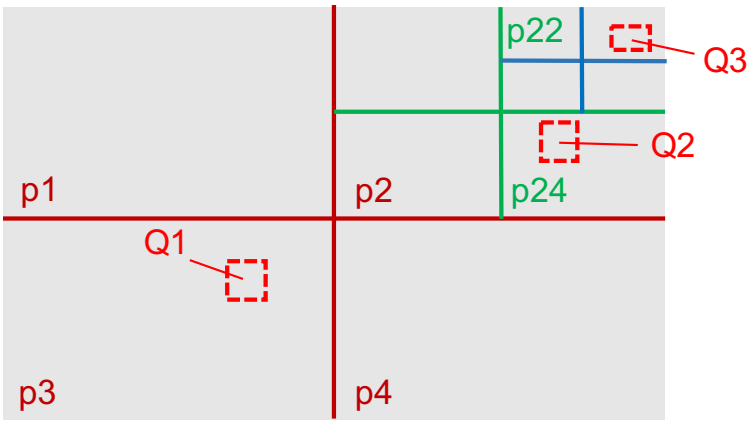


Figure 5.2 – Incremental indexing strategy (in 2D).

5.3.1 Refinement Concept

More precisely, Space Odyssey incrementally builds an Octree [33] on each dataset queried. The Octree is the index of choice since we want to introduce minimal overhead during the query execution and thus, we split each dimension to a minimal number of partitions which corresponds to 2^d partitions in d-dimensional space. Figure 5.2 illustrates the indexing process with $d = 2$, i.e., 4 partitions per level. The indexing process starts with the first query Q1 where Space Odyssey partitions the space uniformly into four partitions (p1, p2, p3, and p4). It scans the dataset and assigns each object to the partitions it overlaps with. When the second query arrives (Q2), Space Odyssey identifies the partitions that it intersects with (only p2 in our example), refines this partition, i.e., divides p2 into four sub-partitions (p21, p22, p23, and p24) and reassigns its objects to the new partitions. In the same process it checks for the objects in the qualifying new partitions whether or not they are inside Q2. Space Odyssey applies the same procedure for query Q3 and all subsequent queries.

Space Odyssey refines partitions to curb the amount of data retrieved and checked for intersection with the query. Otherwise, entire massive partitions need to be checked even only for a small query. Intuitively, we want the partition size to approximate the query size. Then, in the best case, a query hits only one partition which covers just the queried range so that a single sequential scan of the partition retrieves all required objects. In the worst case, 2^d partitions are intersected by the query. Refining a partition further only incurs unnecessary processing overhead (the actual refining as well as retrieving and scanning multiple resulting partitions). Therefore, to control the degree of refinement Space Odyssey uses a *refinement threshold* (rt). A partition is refined following the execution of a query if the ratio $\frac{V_p}{V_q} > rt$ where V_p and V_q are partition and query volumes, respectively.

With this incremental refinement strategy the overhead of building the index and reorganizing the data on disk is spread over several queries. Areas frequently queried will be indexed fully, i.e., very fine granular such that range queries in these areas can be executed efficiently, as

efficient as if executed on a fully built Octree. Areas previously untouched will be partitioned at a coarser granularity thus queries in these areas can also benefit from the adaptive partitioning performed due to previous queries.

5.3.2 Optimizations

In case the query size is significantly smaller than the partition currently hit, it might require a considerable number of queries until the partition is refined enough. We can compute by how many queries a partition needs to be hit before it reaches the finest level of refinement (or put differently, before the Octree reaches the targeted depth) with the refinement threshold. The following equation gives the number of queries (or levels in the Octree built) required:

$$\log_{ppl}(V_p/(V_q \times rt))$$

where ppl is the number of partitions per level and $ppl = 2^d$ in a standard Octree. To allow for faster convergence we can set Space Odyssey to use a bigger ppl.

Since we use space-oriented partitioning a spatial object can intersect with several partitions which introduces additional intersection tests. To avoid object replication and thus curb the memory footprint while avoiding unnecessary comparisons, we translate the problem of indexing volumetric objects to indexing point objects by using the query window extension technique [122]. Space Odyssey assigns each object o to a partition based on o's center and keeps track of the maximum object extent (*maxExtent*) in each dimension. Then, to answer a query correctly ensuring that all intersecting objects are retrieved, its range is extended by *maxExtent* and the cells the extended query overlaps with are inspected.

Finally, Space Odyssey performs the updates in-place, i.e., it reads a partition p, refines it and uses the pages where partition p was stored for the newly created partitions. After refining p we may require more disk pages than were initially required to store p; we append these pages at the end of the file.

5.4 Combining Datasets

By building data structures incrementally we can significantly decrease the data-to-insight time. At the same time we have the opportunity to optimize the placement of data structures on disk to accelerate the queries executed.

Particularly in the case where multiple datasets are analysed, apart from building an index structure incrementally for each dataset, Space Odyssey also rearranges the data on disk such that the areas in different datasets which are queried together are also stored together. Doing so allows Space Odyssey to avoid random disk access for retrieving the same area in different files and thereby accelerates access.

5.4.1 Merging Partitions

While executing queries Space Odyssey keeps statistics about the datasets queried together and the partitions retrieved from them. More precisely, given queries of the form $Q = \{A; DS_1, \ldots, DS_N\}$ where A is the area queried in datasets DS_1 through DS_N, it will store: 1) how often a given combination $C = \{DS_1, \ldots, DS_N\}$ is accessed and 2) what partitions are retrieved from C, i.e., what partitions P overlap with A.

Once the number of retrievals for a particular combination C of datasets exceeds a preset *merging threshold* (*mt*), Space Odyssey merges the data for all the partitions $p \in P$ retrieved in the context of C. It iterates over all partitions that have been queried for in C, retrieves them from every dataset $DS \in C$ and merges them on disk. Note that some of the merged partitions may be retrieved less frequently by past queries than others, but the overhead of including them in the merged file is minimal while there is a benefit in case they are accessed more frequently in the future. Lastly, Space Odyssey merges data only for combinations of size $|C| \geq 3$ because merging is more beneficial for bigger combinations as it prevents (random) accesses to a large number of datasets.

5.4.2 Data Structures

Space Odyssey creates a new *merge file* where it stores the partitions P from different datasets queried together in a combination C so they can be read sequentially and hence more efficiently once they are again queried together. The partitions in the merge file are copies, meaning that Space Odyssey also keeps the original partitions to support efficient querying on an individual dataset DS. Space Odyssey maintains a space budget for merge files (and thus replicated partitions). Once the space budget is exceeded it removes the least recently used merge files to adhere to the budget.

For a given partition p, the merge file physically stores the objects contained in p from each dataset DS sequentially. Given, for example, datasets DS_x, DS_y, DS_z, Space Odyssey stores objects from DS_x on the first disk pages, followed by objects from DS_y followed by DS_z. Doing so allows to retrieve efficiently only the objects belonging to a queried subset of all datasets merged (e.g. DS_x and DS_z) by reading them sequentially while skipping over the rest (DS_y). The merge file is append-only, i.e. new partitions are always added at the end of the file.

Space Odyssey incrementally builds index structures per dataset and the same regions in different datasets may thus have a different level of refinement. In dataset DS_x, for example, the area may still only be covered by one partition p while it is divided into eight partitions in DS_y. Including copies of the unrefined partition p in merge files adds the challenge of having to refine all the copies once refinement of p is triggered by a new query, thereby introducing substantial overhead. Space Odyssey addresses this issue by only merging partitions which are at the same level of refinement. Additionally, in our current implementation the merged partitions are not refined any further.

5.4.3 Querying

To efficiently execute queries and take advantage of merge files, i.e., to decide whether to retrieve areas from individual datasets DS or from merge files, Space Odyssey maintains a directory where it keeps information about what partitions of what combinations of datasets are stored together.

Once a query $Q = \{A; DS_1, \ldots, DS_N\}$ is to be executed, Space Odyssey checks what partitions intersect with A and whether these partitions are stored in a *merge file*. There are four possibilities:

Exact Merge File. If the exact combination $C_q = \{DS_1, \ldots, DS_N\}$ is stored in a merge file and contains the partitions intersecting with A, then it is used to retrieve those partitions sequentially.

Superset. If a superset $C \supset C_q$ is stored, i.e. the merge file contains more datasets than the ones requested, then the merge file will still be used. Using the merge file is more efficient than accessing individual datasets thanks to the internal organization of merge files: the objects from each dataset are organized sequentially, meaning that they can be read efficiently but also that if data from a particular dataset is not needed it can be skipped.

Subset. If a subset $C \subset C_q$ is stored, i.e. the merge file contains fewer datasets than the ones requested, then Space Odyssey uses the merge file to retrieve all data from the subset C as well as other merge files or individual files to retrieve the remaining datasets $C_q \setminus C$. The decision which of the merge files to use is based on maximizing the number of datasets already stored in a merge file and thus minimize (random) access to individual files. Space Odyssey chooses the one merge file which contains the most datasets queried for.

No Merge File. If no merge file exists for a combination C, individual files are used.

5.4.4 Open Issues

Building a merged index for the hot areas where multiple data sets are queried together significantly accelerates queries but several challenges need to be addressed to fully automate the merging and maximize the performance gains. In particular, we plan to develop a cost model which indicates how to adapt the parameters (minimum size of combination to be merged $|C|$ and mt) at runtime based on the workload. Additionally, we plan to investigate the benefits of merging partitions at different refinement levels and examine alternative strategies for doing so, e.g., should all partitions be refined to the same level as the finest partition before merging or as the coarsest, or shall we allow multiple refinement levels to coexist in the merged index. Lastly, we plan to improve disk space management to avoid the replication of a dataset which is used in several different combinations whenever possible.

5.5 Experimental Evaluation

In this section we first describe the experimental setup and methodology and then demonstrate the behavior of Space Odyssey by comparing it against state-of-the-art spatial indexing approaches using real neuroscience datasets.

5.5.1 Experimental Setup

Hardware Configuration. The experiments are run on a Linux Ubuntu 12.04 machine equipped with 2x Intel Xeon Processors, each with 6 cores running at 2.8GHz, with 64kb L1, 256KB L2 and 12MB L3 caches and 48GB RAM at 1333MHz. Storage consists of 2 SAS disks of 300GB capacity each.

Competing Approaches. We have implemented Space Odyssey and set its configuration parameters $rt = 4$, $ppl = 64$, and $mt = 2$. Additionally, we consider the following approaches:

FLAT is the state-of-the-art indexing technique for spatial range queries, for which we obtained the source code from the authors [127]. As we need to index multiple spatial datasets, we implement two strategies: one-for-each (1fE) and all-in-one (Ain1). The first strategy, 1fE, builds one index for each dataset. To perform a query, all the indexes corresponding to the queried datasets are probed and the union of the retrieved results forms the final answer. The second strategy, Ain1, builds only one index structure containing all the spatial objects from all the datasets. To perform a query, the index is probed and items belonging to datasets which are not queried are filtered.

Grid is a static, uniform grid-based technique where the indexed space is uniformly partitioned into a fixed number of cells. We use our own implementation. The objects are assigned to the grid cells in-memory and flushed to disk when the memory buffer becomes full. Similarly to Space Odyssey, replicating objects to multiple grid cells is avoided by using the query window extension technique [122]. The configuration is set to 60^3 cells, which we determine through a parameter sweep, given the absence of heuristics. We use 1fE as a default strategy for Grid given that both strategies, 1fE and Ain1, achieve similar performance as a result of the employed space-oriented partitioning.

Software Setup. All implementations are written in C++, they are single-threaded and compiled using g++ v4.9.2 with the `-O3` optimization flag. The disk page size is set to 4KB. To obtain realistic run-times, where dataset sizes are significantly larger than the main memory size, all techniques are restricted to 1GB of main memory footprint. For all experiments only one disk is used (i.e., no RAID configuration) while the OS caches and disk buffers are cleared (overwritten with an empty file) before each query is executed to avoid any caching effects.

Datasets. We use 10 real neuroscience datasets that we obtained from our collaboration with neuroscientists in the Human Brain Project [82]. Each dataset represents a subset of neurons contained in the same brain volume. The neurons are modeled with a 3D surface mesh. An

identifier is attached to each object to distinguish items belonging to different datasets. Each dataset requires approximately 5 GBs of storage on disk (and ~ 50 GBs in total).

Queries. Based on the previously described use cases, we synthetically generate queries each having a fixed volume (*qvol*) of 10^{-4}% of the queried brain volume. We use a *clustered* distribution and choose a number of *clustercenters* ($|clusterscenters| = 10$). Query centers are distributed around the cluster centers following a Gaussian distribution ($\mu = 0$, $\sigma = qvol \times 10$). For completeness and to evaluate non-skewed cases, we also generate uniformly distributed query centers. We generate a workload of 1000 queries.

To choose which subset of datasets is queried for each query range, we use a synthetic distribution generator based on Gray et al. [40]. The distributions we use are: (1) heavy hitter, (2) self-similar, (3) Zipf, and (4) uniform. These distributions have been used in other studies for similar purposes (e.g., in [20, 117]). In the heavy hitter distribution, one combination of queried datasets accounts for 50% of all possible combinations, while the other queried combinations are chosen uniformly from the remaining ones. The self-similar distribution uses an 80–20 proportion, and the Zipf distribution uses an exponent of 2. For non-skewed scenarios, we also choose the combination of datasets randomly using a uniform distribution.

5.5.2 Experimental Analysis

Total Processing Cost. Figures 5.3, 5.4, and 5.5 depict the total workload processing time when queries follow a clustered distribution, while the subsets of the dataset queried are chosen according to Zipf, heavy hitter, and self-similar distributions respectively. In Figure 5.6, we uniformly choose both the query ranges, as well as the queried datasets, in order to demonstrate the worst-case performance, where neither hot areas nor popular combinations exist. In all experiments the number of queried datasets is increased from 1 to 9 (note that while the number of possible combinations to query increases from 10 and peaks at 252, the actually queried combinations are often smaller and depend on the distribution; also shown on the x axis). For Space Odyssey's competitors, the processing time is additionally broken down into indexing and querying.

FLAT is based on data-oriented partitioning and therefore, the FLAT variants are the slowest to build among all approaches. Indexing with FLAT is up to ×5 slower compared to the simple uniform Grid[2]. As such, only Grid is competitive in terms of overall data-to-insight time when compared to Space Odyssey. Nevertheless, by the time Grid finishes indexing the data, Space Odyssey has already answered half of the queries on average. Therefore, the important aspect of Space Odyssey is that it minimizes data-to-insight time, because there is no need to build complete indexes for all the datasets in advance.

[2] Favouring Grid, we assume that the optimal configuration is known. Otherwise, several builds of Grid are required to tune it.

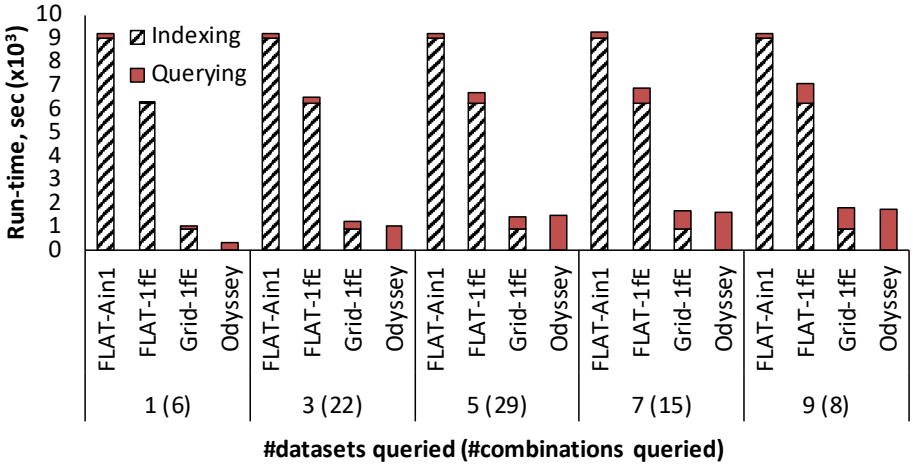


Figure 5.3 – Query ranges: clustered, dataset ids: zipf.

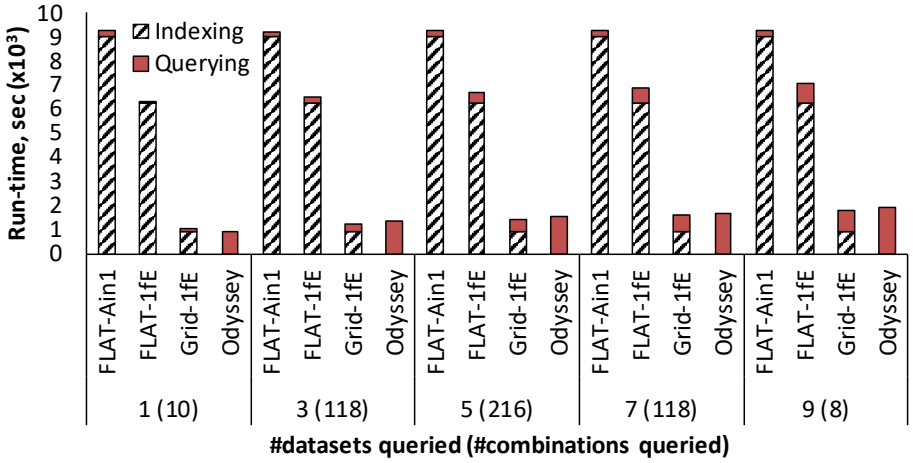


Figure 5.4 – Query ranges: clustered, dataset ids: heavy-hitter.

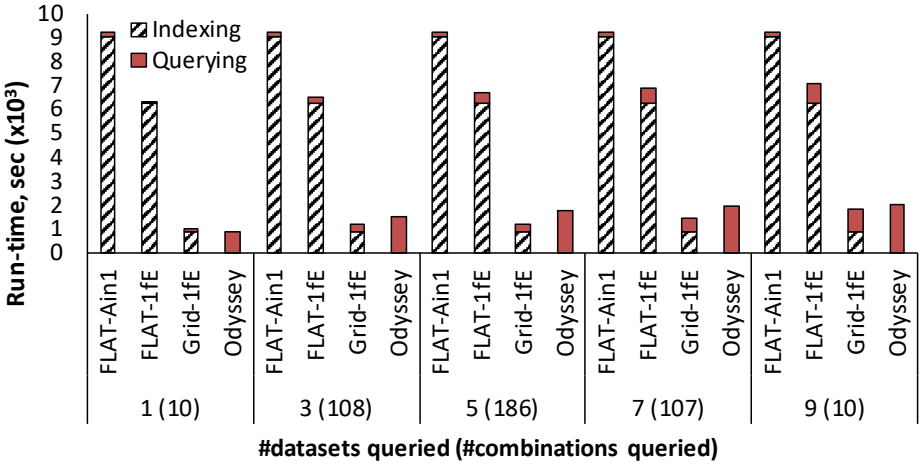


Figure 5.5 – Query ranges: clustered, dataset ids: self-similar.

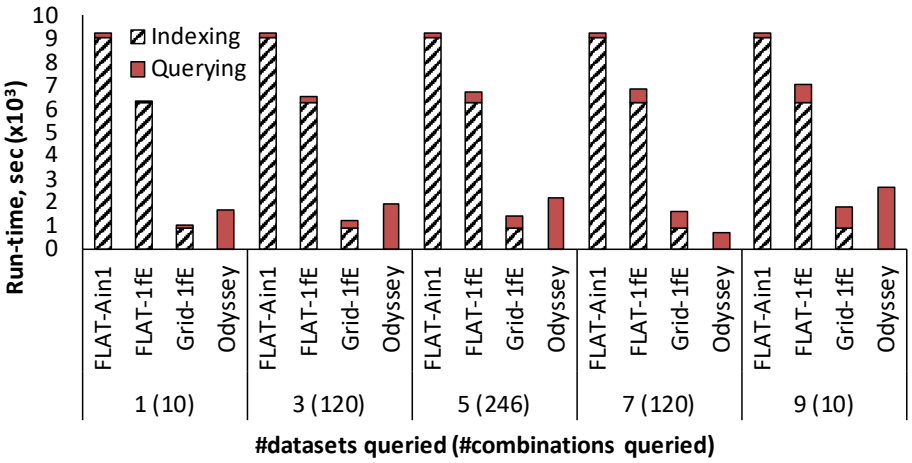


Figure 5.6 – Query ranges: uniform, dataset ids: uniform.

Space Odyssey is a middle ground between the one-for-each and the all-in-one strategies. The one-for-each (1fE) strategy accesses individual (smaller) indexes and only for the datasets queried. Consequently, the query processing cost increases with the number of queried datasets. The all-in-one (Ain1) strategy, on the other hand, always operates on a huge index structure and suffers from unnecessary data accesses. As such, when the number of queried datasets is less than 5, 1fE is preferred over Ain1. Space Odyssey follows a hybrid strategy, where the individual datasets are indexed adaptively (similarly to 1fE), but hot areas from different datasets are merged together (similarly to Ain1).

While all related approaches are insensitive to skew in the workload, the adaptive mechanisms in Space Odyssey are able to exploit it. For example, when the queried dataset combinations are coming from the very skewed zipf (Figure 5.3) and heavy-hitter (Figure 5.4) distributions, Space Odyssey quickly refines the hot areas, merges the partitions of the popular datasets together, and is often able to perform most of the queries before even Grid finishes building. This is not the case with the less skewed self-similar distributions (Figure 5.5), where Grid (once its building phase is over) answers individual queries faster than Space Odyssey most of the time. When both query ranges and queried datasets are uniformly distributed (Figure 5.6), Space Odyssey cannot benefit from adaptive refining and thus requires more time than Grid to process the entire workload of 1000 queries.

Query Performance. In Figure 5.7, we show the response time for each query in the sequence when 5 datasets are queried. In Figure 5.7a, the queries are clustered and the queried combinations of datasets are chosen from the self-similar distribution, while in Figure 5.7b both the queries and the combinations are chosen from a uniform distribution. We study Space Odyssey and the two approaches that are the most competitive in terms of querying performance: FLAT-Ain1 and Grid-1fE. In both cases, the very first query is the most expensive for Space Odyssey, as it fully scans and partitions at the first (coarsest) level the raw data files for all 5 datasets in the combination. Nevertheless, we observe that Space Odyssey converges to

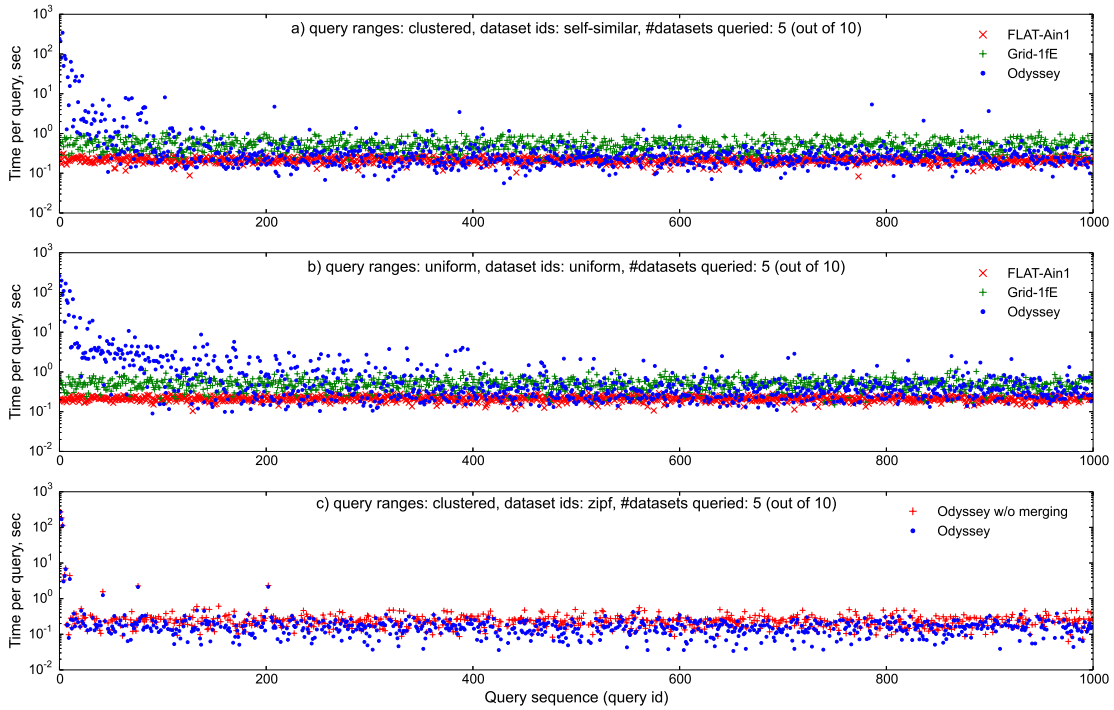


Figure 5.7 – Query times for each query in a sequence.

the speed of the fully indexed case under both skewed (Figure 5.7a) and uniform (Figure 5.7b) scenarios. As expected, the convergence is slower in the uniform scenario. FLAT-Ain1 has consistently better and more robust performance than Grid-1fE because it is less sensitive to data skew. Once Space Odyssey has converged, its querying performance is between FLAT-Ain1 and Grid-1fE, while it performs some queries even faster than FLAT-Ain1 as a result of the merging strategy. Finally, when an area that has not been previously refined and/or merged is queried, the querying time for Space Odyssey is still higher.

Effect of Merging. Lastly, to isolate the effect of merging partitions that are often queried together, we run Space Odyssey with and without merging enabled. In this experiment, clustered queries are produced using 5 instead of 10 *clustercenters*, in order to ensure that the queries can benefit from merging. In Figure 5.7c we plot the execution times only for the queries that request the most popular combination (for the Zipf distribution, this combination is queried 751 times). While the same combination may still request completely different ranges (e.g., in different clusters), we see that eventually Space Odyssey benefits from the merged partitions for the majority of the queries. We observe 25% performance gain on average for the queries accessing the merged partitions.

5.6 Conclusions

In this chapter we identified the challenge of efficiently exploring multiple spatial datasets with the same range query – a common type of analysis across scientific applications. State-of-the-art methods fall short in supporting this challenge efficiently, as they require to index *all* data a priori, including the parts never analyzed.

As a consequence, we develop Space Odyssey, an approach which incrementally indexes the bits of the data needed and adapts the physical layout of the data on disk to efficiently support the queries executed. Our approach to incrementally indexing and reorganizing spatial data on disk shows benefits in decreasing the data-to-insight time. Although the current implementation of Space Odyssey already achieves speedup, we primarily consider it a starting point demonstrating the potential of our idea. In particular, we believe that refining the cost model for merging and indexing can further increase performance benefits.

6 In-Memory Incremental Indexing

With large-scale simulations of increasingly detailed models and improvement of data acquisition technologies, massive amounts of data are easily and quickly created and collected. Traditional systems require indexes to be built before analytic queries can be executed efficiently. Such an indexing step requires substantial computing resources and introduces a considerable and growing data-to-insight gap where scientists need to wait before they can perform any analysis. Moreover, scientists often only use a small fraction of the data – the parts containing interesting phenomena – and indexing it fully does not always pay off.

In this chapter we present a novel incremental index for the exploration of spatial data. Our approach, QUASII[1], builds a data-oriented index as a side-effect of query execution. QUASII distributes the cost of indexing across all queries, while building the index structure only for the subset of data queried. It reduces data-to-insight time and curbs the cost of incremental indexing by gradually and partially sorting the data, while producing a data-oriented hierarchical structure at the same time. As our experiments show, QUASII reduces the data-to-insight time by up to a factor of 11.4x, while its performance converges to that of the state-of-the-art static indexes.

6.1 Introduction

The advances in data acquisition technologies and supercomputing for large-scale simulations rapidly increase the amounts of spatial data generated and collected. For instance, in the Human Brain Project (HBP) [82], neuroscientists build spatial models of the brain which will ultimately feature 10^{11} neurons [127], each reconstructed with thousands of $3d$ cylinders. NASA released 500 TB of earth observation data generated through remote sensing [87], while the Dutch government released point cloud data with 640 billion points [88] acquired through airborne scanning. Similarly, volunteers generate large amounts of spatial data through services such as OpenStreetMap [94]. Given these massive and growing amounts of spatial data, algorithms to query them efficiently are crucial.

[1] QUASII originally appeared in [103].

Previous research has proposed many techniques [33, 70, 127] for the fast and scalable querying of spatial datasets. Existing approaches, however, have two major drawbacks. First, they require a time-consuming step to build indexes before they can be used. This pre-processing step significantly delays the analyses: indexing a model in the HBP, for example, can take several hours [127]. With increasing dataset size, the data-to-insight time grows as well. Second, scientists frequently only analyse a small fraction of the data [2, 140]. In the HBP, for example, a scientist builds a model of the brain but after a few queries may determine that it is not biorealistic (e.g., density in certain areas does not agree with measurements) and stops the analysis. Given the small number of queries executed, the overhead of indexing the entire model cannot be fully amortized.

The problems of delayed analysis (due to prior indexing) and the impossibility to amortize indexing cost (due to too few queries) are not exclusive to spatial data management. Database research has proposed incremental indexes for relational data (e.g., cracking [53] and adaptive merging [38]) and for time-series [140]. The core idea is to incrementally index only the parts of the data queried, spreading the cost of indexing over the first few queries. The major data-to-insight bottleneck is thus eliminated, i.e., queries are answered as soon as data is available (albeit the first queries run slower, as no index is initially available).

To address the aforementioned challenges, we develop an incremental indexing approach for spatial data in main memory, with the aim of reducing data-to-insight time, as well as achieving performance comparable to traditional spatial indexes (after enough queries are executed). As no current incremental indexing approach for main memory exists, we demonstrate the limitations of applying current options to incrementally index spatial data. As we show, using the concepts for incrementally indexing one-dimensional data [53] to index three-dimensional data does not significantly reduce data-to-insight time, as the major bulk of work still has to be done for the first query. Space Odyssey [101] (Section 5) is designed for exploratory analyses of multiple spatial datasets that reside on disk. Its main benefit comes from indexing only the subsets of multiple datasets and optimizing accesses on disk to the parts of the datasets frequently queried together. However, in a different setting, indexing of one dataset in main memory, these benefits disappear.

We thus develop a QUery-Aware Spatial Incremental Index - QUASII: a novel data-oriented, query-driven incremental indexing approach. QUASII substantially reduces data-to-insight time and keeps the cost of incremental strategy low, by gradually and partially sorting the spatial objects considering all dimensions. QUASII thus distributes the cost of indexing across all queries, while preserving spatial proximity and producing a data-oriented style partitioning – which typically entails an expensive pre-processing step in the static setting. Finally, being data-oriented, it executes queries efficiently, as it adjusts to the distribution of the data, while avoiding data replication.

Our experiments show that QUASII substantially accelerates the exploratory analysis of spatial data in main memory by reducing the data-to-insight time by up 11.4×, while achieving the

query performance of current algorithms for spatial indexing. Static algorithms are not able to amortize their building cost over QUASII even after 10000 queries.

To our knowledge we are the first to develop and analyze incremental indexing for spatial data. Our contributions are:

- We demonstrate the challenges of adapting and using known incremental indexing [53, 101] to spatial data in main memory. We use the resulting approaches as motivation and baseline.
- We develop *QUASII*, an incremental approach that significantly reduces the data-to-insight time, while achieving the query performance of state-of-the-art spatial indexes.
- We experimentally analyse QUASII's performance and the number of queries it needs to reach the performance of its static counterparts.

The remainder of the chapter is structured as follows. We define the problem in Section 6.2 and motivate it in Section 6.3. We then describe QUASII in Sections 6.4 and 6.5 and experimentally evaluate it in Section 6.6. We conclude in Section 6.7.

6.2 Problem Definition

Our work is driven by the need for the exploratory analysis of spatial datasets through querying. The queries executed are ad hoc, i.e., the next query is only known after the results of the first query are analyzed, and they thus cannot be batched and executed with only one sequential read of the dataset.

Example Application. In the Human Brain Project, neuroscientists build spatial models of the brain [82]. Already now the models are so detailed that to simulate a neocortical volume of only 0.29 mm^3 supercomputers are needed [83].

Once the part of a model is built, neuroscientists need to validate it by choosing a subset of its regions at random and inspecting them. Each region is queried with several spatially close queries and the query results are used to verify the composition, density and other metrics agree with the real brain. The results of these analyses are crucial to determine whether or not the model can be simulated or should be abandoned (subsequently building a new one using a different configuration). Scientists currently only have two fundamentally different options: index all data a priori and execute queries with the index or scan all data each time to answer a query. Not knowing a priori how many queries will be executed (and if indexing can be amortized) makes it difficult to decide.

Data. We consider spatially extended (volumetric) objects enclosed by a minimum bounding box (MBB). In a three-dimensional ($3d$) setting, each MBB b is defined by two $3d$ points $lower(b)$ and $upper(b)$ corresponding to lower and upper coordinate at each dimension ($lower(b) = (x_l, y_l, z_l)$ and $upper(b) = (x_u, y_u, z_u)$) [33].

Queries. We focus on range (window) queries as they are broadly used in many applications and are also the building block for many other spatial queries (e.g., k-nearest neighbor queries [60]). Each query is a $3d$ box specified by two $3d$ points, e.g., (q_l, q_u). Given a query q, all objects with their bounding box b intersecting with q, i.e., where $b \cap q \neq \emptyset$, are in the result.

Setting. We assume that all data and necessary index structures fit in main memory. We consider a static setting, i.e., all raw data is available before querying.

6.3 Motivation

No current incremental indexing approach can index spatial data in main memory. To explore the possibility of using relational incremental indexing approaches to index spatial data, we extend database cracking [53] to spatial domain. We also adapt our approach, Space Odyssey (introduced in Chapter 5), for main memory when indexing just one spatial dataset.

6.3.1 Cracking for Spatial Data

Relational Cracking. Database cracking [45, 53, 54] incrementally builds an index as a byproduct of query execution in the context of main memory column-stores. The proposed techniques partially sort elements based on the query execution, essentially performing an incremental quick sort. In its simplest form, cracking [53] rearranges elements in an array according to the end points of the query range (q_l, q_u): all values $< q_l$ are moved towards the beginning of the array, while values $> q_u$ are moved towards the end. With each query, the index becomes more refined until it is fully sorted and indexed.

SFCracker. Using this strategy to index spatial data is inherently challenging: spatial data has multiple dimensions and, unlike $1d$ data, no total order can be directly imposed on it. Therefore, to be able to use the strategy of cracking we transform data from the multi- to the one-dimensional domain. We perform this transformation using a space-filling curve (SFC) – a common approach to impose a total, $1d$ order on spatial objects.

A SFC maps data to $1d$ domain by visiting all the points in a d-dimensional grid exactly once; the order in which the objects are visited defines their order in $1d$ space. When mapping spatial data, it is crucial to consider SFCs that preserve proximity (such as Z-order [96] or the Hilbert curve [59]), so that data points close in multi-dimensional space remain close in $1d$ space [29, 86].

The resulting approach, *SFCracker*, incrementally sorts SFC codes based on the queried region. Both, data and queries are transformed to $1d$ space. The data transformation takes place in the first query, which makes it the most expensive one. Once the data is transformed, the queries perform cracking based on the $1d$ intervals obtained through the query transformation.

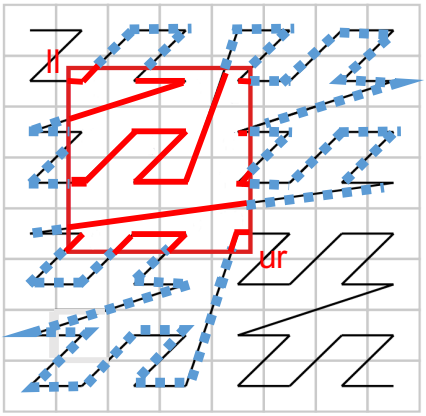


Figure 6.1 – Overhead introduced when transforming query to the 1D space.

A naive query transformation to $1d$ space results in a substantial number of false positives (needed to be tested for intersection) because the transformed $1d$ range can be significantly larger than the original multidimensional range if only the *lower* and *upper* coordinates of the range query are considered. An example is shown in Figure 6.1: the curve segments in blue belong to the transformed range $(SFCcode_l, SFCcode_u)$, but they are outside of the original query range (in red). To reduce the overhead of false positives, we use a technique that partitions the curve into multiple sub-intervals each of which is fully contained in the original range [132]. Consequently, a range query is transformed into a number of intervals and the data is thus cracked multiple times per query, once for every interval.

Limitations. Cracking in the relational domain decreases data-to-insight time, distributing the cost of sorting over all queries with fairly low overhead and initialization cost. These benefits, however, decrease for datasets with a higher number of dimensions. First, the initial query is expensive as it maps all the objects from the multi- to the one-dimensional domain. Second, as opposed to relational data, a single query has to perform multiple expensive cracks to avoid performance penalties introduced with the transformation to $1d$ space. Consequently, spatial cracking still has a considerable data-to-insight time, along with an expensive incremental strategy. We demonstrate these limitations experimentally in Section 6.6.3.

6.3.2 Disk-based Incremental Indexing in Main Memory

Disk-based Incremental Indexing. Space Odyssey (introduced in Section 5) is designed for exploratory analyses of multiple spatial datasets that reside on disk. Although Space Odyssey addresses a different problem, we use its ideas related to incremental indexing, to explore its applicability and limitations when indexing a dataset in main memory. We adapt them for use in main memory in *Mosaic*.

Mosaic. Mosaic incrementally builds an Octree [57] by dividing the space uniformly into eight partitions. For every query, Mosaic identifies the partitions that overlap with the query,

splits them into eight partitions and reassigns their objects to the newly created partitions. Frequently queried areas in a dataset are indexed fully, whereas less frequently queried areas are coarser grained. The top-down strategy is thus beneficial for consecutive queries, as they can reuse the previous partitioning, independent of the workload pattern. However, data in frequently queried areas is re-partitioned multiple times.

Limitations. The main benefit of Space Odyssey comes from indexing only the subsets of multiple datasets (as opposed to indexing all datasets upfront) and optimizing accesses on disk to the parts of the datasets frequently queried together. However, in a different setting, indexing of one dataset in main memory, these benefits disappear. More precisely, Mosaic introduces significant overhead as the data in frequently queried areas is re-partitioned multiple times until it reaches its final configuration. Consequently, a static approach based on space-oriented partitioning, such as the uniform grid, outperforms quickly Mosaic in terms of total execution time (we provide more details in Section 6.6.3).

Mosaic additionally suffers from considering more objects than strictly necessary – a problem inherent in space-oriented partitioning and related to data assignment. For indexes based on space-oriented partitioning, objects can be assigned to cells with two strategies: replication and query extension. Replication assigns an object to all partitions that it overlaps with. As a consequence more objects need to be considered for intersection, the memory footprint increases and an expensive de-duplication step is needed. The alternative is to use query extension [122] which assigns an object to a cell based only on its center. This technique avoids object replication, however, it can considerably increase the number of objects necessary to be tested for intersection. More precisely, to ensure the correctness of the query result, it extends the query range by the maximum object extent. As a result, the area queried for is bigger than the initial query. Both strategies, replication and query extension, slow down query execution but, as we show in Section 6.6.2, replication is particularly expensive when working with volumetric spatial objects and we thus use query extension in Mosaic.

6.4 QUASII Overview

As discussed, an approach to incrementally index spatial data is not as straightforward as adapting known approaches. Besides the challenges, we also identify important design goals:

(i) ***minimal data-to-insight time:*** the main requirement for incremental indexing is to enable instant access to the data, i.e., the first queries must not introduce undue overhead/processing;
(ii) ***efficient query performance:*** the performance of frequently queried subsets of data should converge to that of the fully indexed approach (or better);
(iii) ***low cost incremental indexing:*** indexing should introduce as little overhead as possible, i.e., its cumulative execution time should only exceed the one of static indexes after as many queries as possible (or not at all).

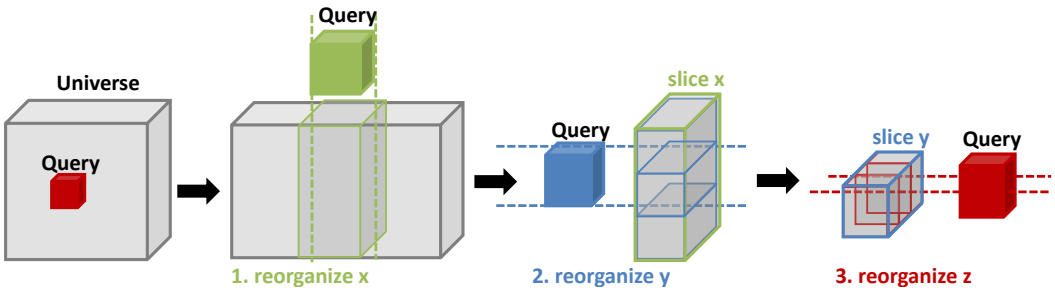


Figure 6.2 – Incremental indexing strategy.

Given the design goals and our analyses, we develop QUery-Aware Spatial, Incremental Index, QUASII. QUASII is a data-oriented index, incrementally built as a side effect of query execution. It reduces data-to-insight time and curbs the cost of incremental indexing by gradually and partially sorting the data, while simultaneously producing a data-oriented hierarchical structure. It is based on a *nested reorganization strategy* which incrementally slices the space in each dimension and a *hierarchical, data-oriented structure* designed to accommodate the incremental indexing process and provide efficient query execution.

Overview. Figure 6.2 illustrates QUASII's incremental strategy on a high level. Given range queries of the form $q = [q_l = (x_l, y_l, z_l), q_u = (x_u, y_u, z_u)]$, QUASII reorganizes the objects based on each query's lower (q_l) and upper (q_u) coordinate by slicing each dimension and performing a nested reorganization. It first reorganizes objects on the x dimension, producing three x slices where the middle one contains the objects in the range $[x_l, x_u]$ given the query range in dimension x. Subsequently, it continues reorganizing the middle x slice on the y dimension, producing again three slices where the middle one contains objects in the range $[y_l, y_u]$. Finally, QUASII reorganizes the y slice on the z dimension producing the z slice which contains the query result. QUASII never performs a complete sort but reorganizes data locally, given the query's boundaries.

The slices produced are organized in a hierarchical structure that incrementally forms the index. Figure 6.3 illustrates the structure of QUASII after the very first query (left) and after an arbitrary number of queries (right) are executed. QUASII forms a hierarchical structure with one level per dimension, i.e., the first (top), second, and third (bottom) levels correspond to slices at x, y, and z dimensions, respectively. The top level has the coarsest granularity as its objects are constrained with one dimension, while the bottom level is the most fine-grained since it is constrained by all dimensions. When executing the queries, QUASII traverses the structure depth-first, performing additional refinements when necessary, as we discuss later in Algorithm 6.

Nested Reorganization Strategy. The incremental strategy of QUASII is query-driven and data-oriented. Being query-driven, it reorganizes the minimal amount of data while executing queries. At the same time, being data-oriented, it achieves query efficiency as it adjusts to the

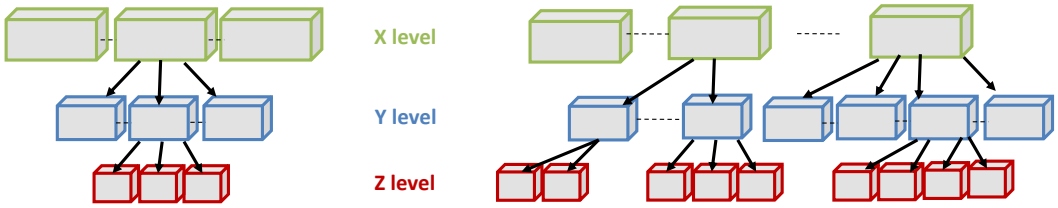


Figure 6.3 – Index structure after the first (left) and few more (right) queries.

data distribution, while avoiding replication. QUASII accomplishes both through its nested reorganization strategy.

Data-oriented partitioning typically entails an expensive pre-processing step in the static setting as it preserves spatial proximity based on a strategy for ordering multi-dimensional objects. QUASII distributes the cost of this pre-processing across all queries by performing nested and partial reorganization. It reorganizes only a subset of data driven by queries, gradually curbing the amount of data partially sorted with every dimension. This strategy is inspired by the Sort-Tile-Recursive (STR) R-tree bulkloading algorithm [70]. STR produces tiles that form leaf-level nodes for the R-Tree by recursively, fully sorting spatial objects in each dimension. More precisely, STR for $3d$ objects first sorts the spatial objects on the x-axis and partitions them in vertical tiles of equal size (i.e., the same number of objects). Then, within each x tile, it recursively applies the same strategy first considering y and then z dimension. This tiling strategy is particularly efficient as the resulting R-Tree has less overlap than other approaches [70]. By only performing partial reorganizations for the parts of the data that is actually queried, QUASII outputs partitions targeting these characteristics at lower cost (as opposed to complete sorts in STR).

Index Structure. QUASII's index structure is designed to support an efficient incremental strategy with as little performance penalty as possible. Its hierarchical structure is designed to accommodate the reorganization strategy: each level corresponds to one (reorganization) dimension and each parent node is represented by its children in a nested form along the dimensions QUASII reorganizes data. We discuss the data structure and how it accommodates incremental indexing in more detail in Section 6.5.1.

Benefits. Ultimately, the design choices behind our approach enable us to achieve the goals we outlined. To reduce data-to-insight time (i), QUASII keeps data in the multi-dimensional, spatial domain. This avoids transforming all data at the very beginning which significantly hurts performance of the first query. Next, to achieve query efficiency (ii), QUASII uses data-oriented partitioning that preserves spatial proximity, adjusts to the distribution of data, and avoids object replication. Finally, to keep the cost of the incremental indexing low (iii), QUASII gradually and partially sorts the data using a nested reorganization strategy.

6.5 Data Structure & Query Processing

In the following, we explain the QUASII index structure and data organization before we proceed with discussing querying and incremental indexing algorithms.

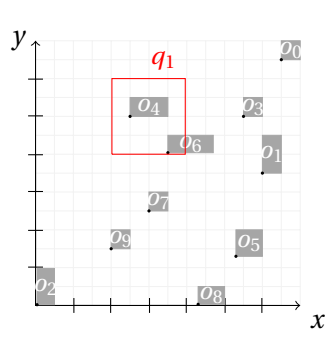
Throughout this section, we refer to a $2d$ example given in Figure 6.4. It depicts a dataset $D = \{o_0, \ldots, o_9\}$ of ten (gray) rectangular spatial objects. All subfigures have three main parts: the top part shows a $2d$ view of the dataset D and how the space is conceptually "sliced" by QUASII, the middle ("Data array") depicts how (raw) data objects are re-organized in main memory, and the bottom shows QUASII's hierarchical data structure that is incrementally built. All x- and y-axis related slicing is marked in green and blue, respectively. Figure 6.4 a) shows the initial state: the "slice-less" view of the data space with D objects and the very first query q_1, the data array of spatial objects in an arbitrary initial order, and the data structure containing the initial slice, s_0 (capturing the entire dataset).

6.5.1 Data Structure

QUASII forms a d-level hierarchical structure, organized according to the number d of dimensions. Each level l has a one-to-one mapping to the corresponding dimension. That is, the first level ($l = 1$) represents slicing of data at x, the second level ($l = 2$) slices at y, and the third level ($l = 3$) slices at the z dimension. The top level always slices data objects at the coarsest granularity, while the bottom level is the most fine-grained. Each slice is described with four attributes: (i) its level, (ii) a minimum bounding box capturing all its objects, (iii) indices to the data array corresponding to the first and last entry of the objects that belong to the slice, and (iv) pointers to sub-slices refining the slice further on the subsequent dimension. In Figure 6.4, this corresponds to the four fields present in each node of the data structure (next to slice label, e.g., s_0): l, box, ids, and arrow pointers (when not null). In our two-dimensional view of the dataset, we mark boxes with a solid line (in the corresponding color), while the slice cuts are marked as dashed lines.

Data-oriented Slicing. One of the main advantages of data-oriented partitioning is that each spatial object is always assigned to just one partition (slice). However, QUASII determines the slices in each dimension based on query ranges. Given volumetric spatial objects, objects can be sliced through and thus overlap with multiple slices. To overcome this problem, QUASII represents each object using only one of its coordinates and uses this coordinate to identify a slice where an object will be assigned to. In particular, during indexing, it uses each object's lower coordinate (x_l, y_l, z_l). Being part of object's MBB, this does not require any additional computation or storage[2]. In Figure 6.4, this coordinate is marked as a black dot for all objects. Figure 6.4 b) illustrates slicing based on the very first query q_1 and its range $[2,4]$ on the x-axis. Slicing at $x = 2$ and $x = 4$ results in three x-slices (s_1, s_2, and s_3). While object o_6 overlaps two slices (s_2 and s_3), it is assigned to s_2 based on its lower coordinate (x_l). Note how the objects

[2] The upper coordinate (x_u, y_u, z_u) or the object's center (requires to be computed, though) can equally be used.



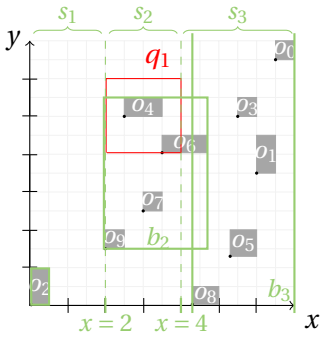
Data array:

o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9	o_0

Data structure:

l=1 s_0
box=inf
ids=0:9

(a) Spatial data and query q_1



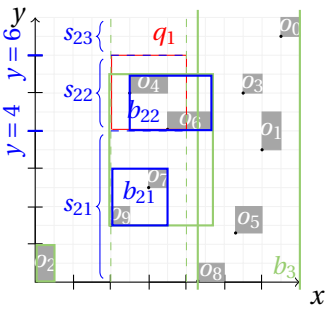
Data array:

o_2	o_4	o_6	o_7	o_9	o_0	o_1	o_3	o_5	o_8

Data structure:

l=1 s_1	l=1 s_2	l=1 s_3
box=o_2	box=b_2	box=b_3
ids=0:0	ids=1:4	ids=5:9

(b) After slicing based on x of q_1



Data array:

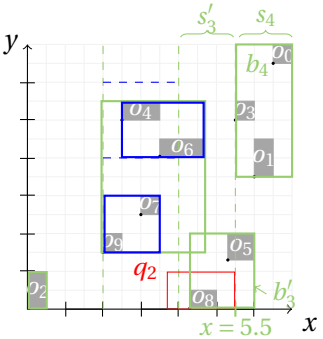
o_2	o_7	o_9	o_4	o_6	o_0	o_1	o_3	o_5	o_8

Data structure:

l=1 s_1	l=1 s_2	l=1 s_3
box=o_2	box=b_2	box=b_3
ids=0:0	ids=1:4	ids=5:9

l=2 s_{21}	l=2 s_{22}
box=b_{21}	box=b_{22}
ids=1:2	ids=3:4

(c) After slicing based on y of q_1



Data array:

o_2	o_7	o_9	o_4	o_6	o_5	o_8	o_0	o_1	o_3

Data structure:

l=1 s_1	l=1 s_2	l=1 s'_3	l=1 s_4
box=o_2	box=b_2	box=b'_3	box=b_4
ids=0:0	ids=1:4	ids=5:6	ids=7:9

l=2 s_{21}	l=2 s_{22}
box=b_{21}	box=b_{22}
ids=1:2	ids=3:4

(d) After processing q_2

Figure 6.4 – An example of query processing and incremental indexing in QUASII (configured with $\tau_x = 4$ and $\tau_y = 2$), given ten spatial objects (o_0–o_9) and two range queries (q_1 and q_2).

are re-organized in the data array and correspond to three partitions (slices) with coordinates $x < 2$, $2 \leq x \leq 4$, and $4 < x$. Accordingly, the data structure is updated with three new (more refined) slices replacing the initial (coarser) slice s_0 (capturing the whole dataset).

While QUASII assigns objects to slices based on their single (lower) coordinate, it records a minimum bounding box for each slice taking into account the actual spatial extent of the objects and thus ensures the correctness of the query result. This also results in slice representations (their MBBs) that are often much smaller but not necessarily within the originally sliced bounds. For example, s_1 contains only one object and thus has a very small MBB (i.e., its $box = o_2$), while the MBB of s_2 is b_2 and exceeds the original cut at $x = 4$ (Figure 6.4 b)). As we show later, this enables QUASII to discard many unnecessary slices during query execution. To limit unnecessary computation (as a slice can be reorganized multiple times until it is fully refined), QUASII computes a full MBB only when a slice is completely refined. Otherwise, a slice is represented with an open-ended MBB, i.e., the MBB has bounds only on the dimension it has been sliced on.

Configuration. QUASII has only one configuration parameter, a size threshold τ, that determines the maximum number of objects in a slice at the finest level. That is, at the bottom level, whenever a slice s contains less or τ number of objects (i.e, $|s| \leq \tau$), it is considered to be fully refined. Intuitively, this is similar to setting a (leaf) node size in the R-Tree.

The sizes of the remaining $d-1$ levels are calculated as follows. Since QUASII performs data-oriented slicing, the total number of partitions required to satisfy threshold τ is $\lceil n/\tau \rceil$, where n is the total number of objects (i.e., $n = |D|$). Consequently, the number of times QUASII has to slice the data space across each dimension to produce $\lceil n/\tau \rceil$ partitions is equal to:

$$r = \left\lceil \sqrt[d]{n/\tau} \right\rceil \tag{6.1}$$

If we use τ_d to denote the slice threshold at the bottom level $l = d$ (i.e., $\tau_d = \tau$), then the maximum number of objects per slice for the remaining levels (up to the top) can be expressed recursively as $\tau_{d-1} = r \times \tau_d$. Note that r corresponds to the number of sub-slices (within a slice) at each index level.

Turning to our $2d$ example[3], after x-based slicing in Figure 6.4 b), s_1 contains one object and thus is considered fully refined (i.e., $|s_1| = 1 \leq \tau_x$), while s_3 has five objects and may be refined in the future. Also note that s_3 stores an open-ended MBB ($s_3.box = b_3$).

The number of levels in QUASII is fixed and always equals to the dimensionality of the queried dataset. That is, it does not depend on the size of the dataset. Therefore, to accommodate the index growth (the index grows in breadth) and enable efficient query execution, QUASII keeps the children (within a slice) organized/sorted according to the level's dimension. QUASII uses this order and the minimum bounding boxes (box) of each node to prune the amount of objects necessary to be tested during the query execution.

[3] To minimize the required number of objects in Figure 6.4, we fix $\tau_x = 4$ and $\tau_y = 2$.

Algorithm 6: query(query q, data D, slices S, result R)

```
1:  S' ← ∅                    // to store newly created (refined) slices
2:  dim ← S[0].l              // current level/dimension of slices in S
3:  i ← binarySearch( S, lower(q[dim]) )
4:  while i < |S| and lower(S[i].box[dim]) ≤ upper(q[dim]) do
5:      if q ∩ S[i].box = ∅ then continue
6:      S'' ← refine( S[i], D, q )      // as per Algorithm 2
7:      for each slice s ∈ S'' do
8:          if q ∩ s.box ≠ ∅ then
9:              if s.l is the bottom level then
10:                 for each j ∈ {s.ids} do
11:                     if D[j] ∩ q ≠ ∅ then
12:                         R ← R ∪ D[j]
13:             else
14:                 if |s.children| = 0 then
15:                     createDefaultChild(s)
16:                 query(q, D, s.children, R)
17:     S' ← S' ∪ S''
18:     i ← i + 1
19: S ← S ∪ S'
20: sort(S)
```

6.5.2 Query Processing and Index Refinement

Having defined QUASII's data structure, we discuss how it is incrementally built and maintained as a side effect of each query.

Query Processing. Algorithm 6 shows the pseudo-code for query processing. Each query traverses the d-level structure depth-first, starting from the first level (having x-slices). Because the slices are sorted, QUASII performs a binary search (Line 3) to find the starting slice. It then scans all the slices $S[i]$ within the query range on the current dimension (i.e., while the loop conditions in Line 4 hold). The loop conditions guarantee that each slice $S[i]$ intersects q only in the current dimension. To discard potential false positive slices early, Line 5 checks if its actual boundaries ($S[i].box$) also intersect with the query range.

Next, QUASII potentially refines $S[i]$ (Line 6), which may be further sliced into multiple more fine-grained slices S'' if it is larger than the maximum size threshold τ (discussed in the next algorithm). In Lines 7—16, QUASII traverses (potentially refined) slices S''. For each $s \in S''$, it either checks all s objects for intersection in case of the bottom level or recursively proceeds querying its children based on the next level/dimension (a default child is assigned to a not fully refined slice, Line 15). Finally, all the newly created slices are accumulated in S' (Line 17), appended to S (Line 19), and re-sorted (Line 20). The slices are sorted based on their ids, i.e., the position (index) of the first slice's object in the data array.

Index Refinement. With each query, QUASII attempts to refine all query intersecting slices (i.e., Line 6 in Algorithm 6). Algorithm 7 provides the simplified pseudo-code for this refinement process. Note that the processing within Algorithm 7 is always based only on the current dimension/level of slice s ($s.l$).

The input slice s is considered for slicing only if it exceeds the threshold τ. Given s is coarse enough, QUASII proceeds with determining the type of slicing based on the intersection between query q and slice s. It considers three types of slicing. If both q's lower and upper coordinates are within s, a three-way slicing is performed splitting s into three sub-slices (Line 5). If only one of q's coordinates is within s, a two-way slicing is performed splitting s into two sub-slices (Line 6). Finally, if q contains s (i.e., both q's coordinates are outside of s's bounds), QUASII performs a two-way slicing based on an artificially introduced coordinate.

QUASII iterates through the generated slices and for the ones that still exceed τ (and overlap with the query) it applies additional refinement according to artificially introduced boundaries in Line 10 (it repeats the process recursively until a slice is fully refined in the corresponding dimension). The three- and two-way slicing algorithms (Line 5 and Line 6) reorganize the data (D) following the incremental quick sort strategy introduced in database cracking [53]. In the reorganization process, QUASII also records the information about the boundaries (box) of newly created or modified slices.

Example. Continuing with our example in Figure 6.4, after refining s_0 into three x sub-slices in Line 6 of Algorithm 6 (and resulting in Figure 6.4 b)), QUASII recursively continues with the intersecting (and just refined) slice s_2 based on the y dimension (Figure 6.4 c)). As such, s_2 is further refined based on the queried y range and results in three new slices (s_{21}, s_{22}, and s_{23}). In this step, only the objects within the s_2 range ($ids = [1..4]$) are three-way sliced and re-organized in the data array. The two new slices (s_{23} is empty) are appended to the data structure as children of s_2. They are fully refined (as $|s_{21}| \leq 2$ and $|s_{22}| \leq 2$) and have much smaller MBBs (b_{21} and b_{22}, respectively) than the initial slice cuts. Finally, because it is the bottom level, the objects within s_{22} are checked against the query range and the two qualifying objects $\{o_4, o_6\}$ are added to the result set (R).

The subsequent query q_2 benefits greatly from previous slicing, as illustrated in Figure 6.4 d). For example, x-slices s_1 and s_2 are skipped completely because query q_2 does not intersect with their MBBs (i.e, test on Line 5 in Algorithm 6). Therefore, QUASII proceeds with the only intersecting slice s_3, which is not fully refined and requires further slicing. As per Algorithm 7, this time a two-way slicing is performed (at $x = 5.5$) resulting in two finer slices (s'_3 and s_4) replacing the previous slice s_3. Next, QUASII continues with y-based slicing of the fully q_2-contained slice s'_3. Since s'_3 reaches the size threshold τ_y, it is not refined further. Finally, the actual data array objects within s'_3 range ($ids = [5..6]$) are checked for intersection with q_2 and the qualifying o_8 is added to the result set.

Artificial Refinement. To produce a balanced hierarchical structure QUASII has to conform with the defined thresholds when forming the slices and using only query boundaries does

Algorithm 7: refine(slice s, data D, query q) $\longrightarrow$ slices S

```
1:  if |s| ≤ τ[s.l] then
        return {s}
2:  S ← ∅                                   // to store refined slices
3:  t ← determineSliceType(s, q)
4:  switch ( t )
5:      case both: S' ← sliceThreeWay(s, q, D)
6:      case one:  S' ← sliceTwoWay(s, q, D)
7:      default:   S' ← sliceArtificial(s, q, D)
8:  for each slice s ∈ S' do
9:      if |s| > τ[s.l] and q[s.l] ∩ s.box[s.l] ≠ ∅ then
10:         S'' ← sliceArtificial(s, q, D)
11:         S ← S ∪ S''
12:     else
13:         S ← S ∪ s
14: return Sa
```

not meet these requirements. One query is usually not sufficient and we cannot use the subsequent queries for this purpose, as they may interfere with the existing order of the slices. For instance, reorganizing a slice again (that has been organized according to all dimensions) based on the x dimension, may disrupt the previously established partitioning for y and z dimensions.

To address this problem, QUASII reorganizes a slice s (Lines 7 and 10 in Algorithm 7) until it meets a size threshold τ in the corresponding dimension. It achieves this by forcing a two-way slicing based on artificially introduced coordinate and thus splitting the slice into two sub-slices. Given the range (x_l, x_u), the new coordinate is $c = \lfloor (x_l + x_u)/2 \rfloor$. The two new slices are recursively sliced further until the threshold τ is satisfied.

While more advanced approaches, e.g., based on the concepts from R*-Tree node splitting algorithms [12], would minimize overlap in data structure, they would also significantly increase the cost of incremental strategy. Therefore, QUASII employs the above uniform and low-cost artificial slicing strategy to meet τ thresholds at each of d levels.

Query & Refine. The outcome of QUASII's reorganization strategy are the slices that are within the query range and consequently only the objects in these slices are checked for intersection. However, performing the reorganization following strictly the query's boundaries would produce an incomplete result, as illustrated in Figure 6.5. For instance, the object o overlaps with the query range q, however, its lower coordinate is outside the query's boundaries and consequently o would not be identified as a part of the result.

To ensure correct query execution while preforming refinement, QUASII employs the query extension technique [122]. More precisely, it extends the query for maximum object extent in

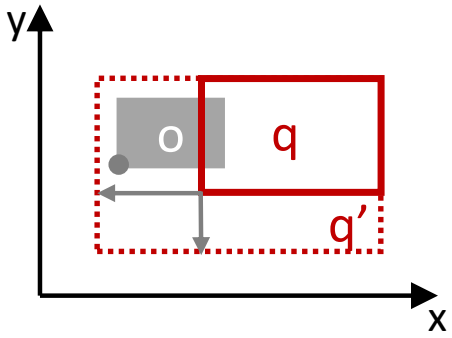


Figure 6.5 – Refinement step: query extension.

each dimension, considering lower coordinate. This extension is done only when performing refinement and only within not fully refined slice. Consequently, the query that performs refinement potentially considers more objects for intersection as its range is enlarged. However, this introduces a minimal overhead as the only alternative is the expensive scan of the entire unrefined slice. We apply the same logic for the binary search where, to avoid missing any slices due to the overlap within them, we extend the query range (while performing binary search) for the maximum slice extent in the corresponding dimension.

6.6 Experimental Evaluation

In this section, we first describe the experimental setup & methodology and then present a thorough experimental analyses that illustrates the benefits of our incremental approach, both on a real-world neuroscience and synthetic datasets. We start the analyses by outlining the shortcomings of the approaches based on space-oriented partitioning in Section 6.6.2. We then study the incremental approaches by comparing them with their static counterparts in Section 6.6.3 and cross-evaluating their performance in Section 6.6.4. Finally, Section 6.6.5 describes the sensitivity analyses of QUASII.

6.6.1 Experimental Setup & Methodology

Hardware. We run our experiments on a Red Hat Enterprise Linux Server release 7.3 machine equipped with 2 Intel Xeon CPU E5-2650L processors at 1.80GHz and 768GB of RAM. Each processor has 12 cores (24 hardware threads) with private L1 (32KB) and L2 (256KB) caches and 30MB of shared L3 cache.

Implementations. All indexing techniques are implemented in C++ and compiled with g++ `4.9.3` with the maximum optimization level. The list below summarizes the implementations that we study:

Scan: performs a full data scan to answer each query.

SFCracker: is our incremental variant of database cracking [53] for spatial data, described in Section 6.3.1. We use the Z-order as a SFC order. The average farthest distance of neighbours in the Z-order is (slightly) higher than in the Hilbert order [29] (i.e., it has better locality), however, we opt to use the Z-order due to its simplicity and the huge body of work on its efficient range query algorithms [11, 120, 132, 134]. We use 32-bit to represent *zcodes* (i.e., 10 bits per dimension) as a trade-off between memory resources and precision (the number of false positives to be filtered).

SFC: is a static counterpart of SFCracker. In the pre-processing phase, SFC transforms data from multi- to one-dimensional domain and sorts it according to the produced SFCcodes. During querying, a $(3d)$ query range is also converted to a $1d$ range and a binary search is used to locate the objects in the $1d$ interval. We employ the same representation of *zcodes* and query optimization as in SFCracker (described in Section 6.3.1).

QUASII: is our incremental approach discussed in Section 6.4. We use 60 objects as a node capacity τ_z.

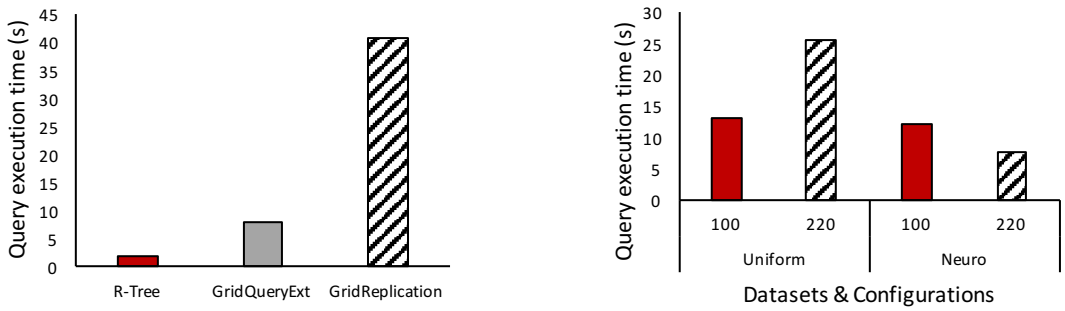
R-Tree: according to our setting, all data is available before querying. Therefore, we use a bulk-loading approach to build the R-Tree index as it reduces overlap and decreases pre-processing time compared to the R-Tree built by inserting one object at a time [70]. For this purpose, we use an efficient STR [70] bulk-loading strategy that balances well the overhead of partitioning the data and query performance. It outperforms Hilbert R-Tree [61] in terms of query performance [70], while its pre-processing cost is not significantly higher [127]. Similarly, TGS [34] and PR-Tree [10] can outperform STR on datasets with extreme skew and aspect ratio, however, they incur considerable overhead for data partitioning. We use the same configuration for node capacity (60) as in QUASII.

Mosaic: corresponds to the space-oriented incremental approach described in Section 6.3.2.

Grid: is a uniform grid-based index, a static counterpart of Mosaic. We use query extension [122] technique (as discussed in Section 6.3.2) to assign an object to a grid cell. We use two configurations with 100 and 220 partitions per dimension for the synthetic and neuroscience datasets, respectively. Both configurations are obtained through a parameter sweep.

Dataset and Queries. We use real-world neuroscience and synthetic datasets.

Neuroscience: we use a small part of the rat brain model represented with 450 million cylinders as elements in a volume of 285 μm^3. We approximate the cylinders with MBBs, resulting in the total number of 450 million MBBs with a size of 21GB on disk. Based on the previously described use cases, we synthetically generate queries, each having a fixed volume *qvol* of 10^{-2}% of the queried brain volume and a clustered distribution. We generate 5 query clusters each with 100 queries, where query centers are distributed around the cluster centers following a Gaussian distribution ($\mu = 0$, $\sigma = qvol$).



(a) Query execution

(b) Configuration

Figure 6.6 – The impact of space-oriented partitioning.

Synthetic: we create synthetic datasets by distributing spatial boxes in a space of 10000 units in each dimension of the $3d$ space. The length of each side of each box is determined uniform randomly between 1 and 10 for 99% of the objects, while 1% of the objects has a side ranging from 10 - 1000 units. The spatial elements are distributed according to a uniform distribution. The datasets have 500 million and 1 billion elements (size on disk 22.5GB and 45GB). For completeness and to test non-skewed cases, we generate *uniform* workload. The uniform workload contains up to 10000 uniformly distributed queries. To have range queries of different selectivity, we vary *qvol*: 10^{-3}%, 10^{-1}%, 1%, and 10% of the universe.

6.6.2 Space-oriented Partitioning Challenges

Both, Mosaic and SFCracker (introduced in Section 6.3), use space-oriented partitioning at their core – Mosaic partitions space, while SFCracker assigns the SFCcodes using a uniform grid. Before we start the analysis of incremental approaches we experimentally demonstrate the shortcoming of space-oriented partitioning – the overhead introduced with data assignment strategy – since it also affects incremental solutions. Further on, we illustrate why a static approach based on space-oriented partitioning, such as a uniform grid, is not a suitable replacement for an incremental index despite having a comparatively cheap pre-processing step (once properly configured).

Data Assignment. In the first experiment we illustrate the impact of data assignment strategies by comparing the performance of Grid and R-Tree. We use two variants of the Grid approach: GridQueryExt avoids the objects replication by using the query extension technique – it assigns an object to the grid partition based on its center, while GridReplication replicates the objects – it assigns an object to all the overlapping partitions.

Figure 6.6a) shows the results of the experiments where we execute 500 clustered queries of selectivity 0.01% on the neuroscience dataset. GridReplication is heavily affected by object replication which increases the number of objects necessary to be checked for intersection and introduces an expensive de-duplication step (needed due to objects replication). GridQueryExt

achieves better performance, however, it still considers 3.1× more objects for intersection than the R-Tree as it extends the initial query for the maximum object extent. The R-Tree clearly outperforms both GridReplication and GridQueryExt with a speedup of 19.4× and 3.7× respectively.

Configuration. In the second experiment we demonstrate the difficulty to configure the grid-based approaches. We use two datasets with identical extent and number of elements but different data distributions: Uniform (uniform distribution, synthetic dataset) and Neuro (skewed distribution, the neuroscience dataset). We use the same experimental setup as for the previous experiment. The best configuration (number of partitions per dimension) is 100 for Uniform and 220 for the Neuro dataset and is determined in a parameter sweep. We measure the execution time when using both configurations for each dataset and illustrate the results in Figure 6.6b).

Although both datasets have the same number of elements and extent, the best configuration significantly depends on the data distribution – the neuroscience dataset requires more partitions compared to the Uniform dataset since it has the very dense regions that require fine grained partitioning. Furthermore, the grid configuration significantly affects performance – the grid performance on the Uniform dataset deteriorates notably when using the Neuro dataset configuration and vice versa.

Summary. Space-oriented partitioning introduces performance penalties. Depending of data assignment strategy, we either consider more elements or suffer from replication. Additionally, the grid configuration is non-trivial and using the wrong one has a detrimental impact on the execution time. In practice we have to use a parameter sweep to find the configuration for a given workload. Consequently, grid configuration turns into a time-consuming process, increasing data-to-insight time.

6.6.3 Incremental versus Static

We first analyze the incremental approaches by comparing their performance with the performance of their static counterparts (introduced in Section 6.6.1). Each static approach has similar properties as its incremental counterpart, however, it involves necessary pre-processing. We categorize the approaches according to these properties as a) one-dimensional, b) space-oriented and c) data-oriented approaches. For each category we present the performance of the incremental approach, its static counterpart and Scan. We first evaluate if and when the approaches converge to the performance of their static counterparts and then analyze the overhead of the incremental strategy. For this purpose we execute the clustered query workload with 500 queries of selectivity 0.01% on the *neuroscience* dataset.

Convergence. In the first experiment we evaluate the convergence of the incremental approaches – how fast an approach converges to the execution time of a fully indexed dataset. Figure 6.7 measures the execution time of each query for all approaches.

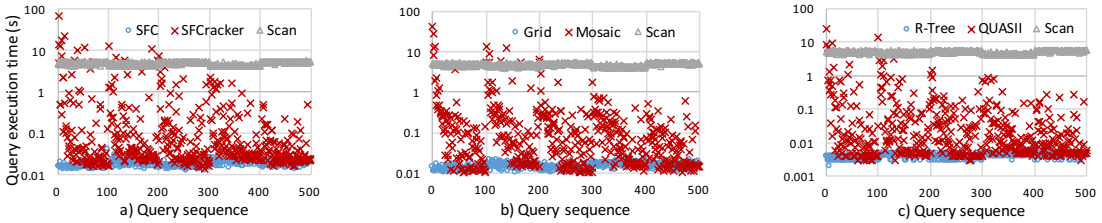


Figure 6.7 – Convergence of a) one-dimensional, b) space-oriented c) data-oriented based approaches.

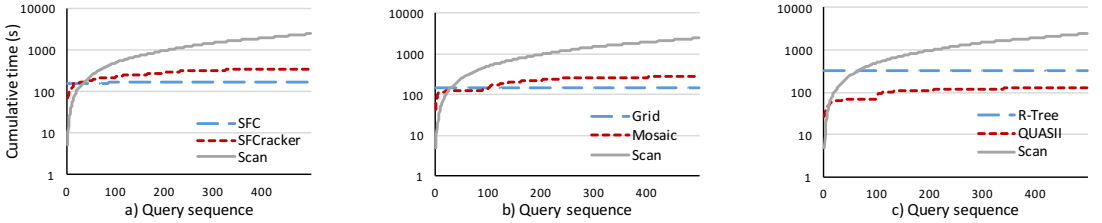


Figure 6.8 – Cumulative time of a) one-dimensional, b) space-oriented c) data-oriented based approaches.

The results show five peaks in execution time, one for each query cluster. The execution of the first cluster of queries (and the associated processing of the data) takes the longest as no index structure exists at the beginning. The first queries therefore exceed the cost of Scan, because at this point, the entire dataset has to be scanned along with building partial index structures. Subsequent queries within a cluster use a partial index and thus execute in less time than a full scan, but take longer than queries on the static approach. This process continues as queries in the same cluster further refine the index. Queries in one cluster not only refine the index locally but also carry out limited, global refinement. The queries in a subsequent cluster thus benefit from previous clusters and execute faster. As the index converges to its full structure, the query execution time approaches that of the to static approach.

Cumulative Response Time. While in the previous set of experiments we measure the individual query performance, in this analysis we measure the cumulative execution time (including index building step for the static approaches). Figure 6.8 illustrates the experimental results.

Similar to the convergence experiment, the query clusters are visible: the cumulative response time jumps each time the experiment moves to a new cluster. The most expensive is the transition from the first to the second cluster while subsequent transitions become less evident as the index becomes more refined.

The cumulative cost of SFCracker is comparatively high and, crucially, with a very expensive first query. One reason is that the first query takes 12.9% of the total pre-processing by assigning the objects to the grid cells and calculating the *zcode* values for the entire dataset.

Adding to this the cost of cracking, the total execution time of the first query grows to 43% of the total pre-processing time. More precisely, in order to minimize the overhead introduced by the transformation to $1d$ space, we partition the $1d$ query range into sub-intervals that tightly cover its original $3d$ range. This optimization [132] results in a high number of small intervals per query – on average 197. As a consequence, the first queries crack the previously uncracked area into a number of small adjacent intervals and therefore reorganize significant amounts of data.

The static (SFC) index, on the other hand, is not substantially slower for the first queries, i.e., the building cost of SFC is not much higher than the first query of SFCracker. In fact, the cumulative execution time of SFCracker exceeds the one of SFC after 23 queries already. The incremental approach SFCracker thus does not offer a considerable benefit over SFC.

The incremental strategy of Mosaic is less expensive compared to SFCracker – the objects within the partition queried are potentially reassigned to the eight newly created partitions based on their location. Therefore, it takes Mosaic longer, i.e., 100 queries, before it exceeds the cumulative time of the static Grid. However, its cumulative execution time is still considerable with the biggest overhead being its top-down incremental strategy. The top-down strategy ensures fast convergence but it also introduces overhead as the data in frequently queried areas is re-partitioned multiple times until Mosaic reaches its final level of refinement.

QUASII, at the same time, does not exceed the cumulative execution of the R-Tree in our experiments. Even after 500 executed queries, the cumulative execution time for QUASII is 39.4% of that of the R-Tree. The main benefit comes from its partial reorganization strategy where the objects are gradually reorganized within the query boundaries, as opposed to the complete sort.

Summary. While all the incremental approaches reach the performance of their static counterparts, the incremental strategies of SFCracker and Mosaic are comparatively expensive. As we show for SFCracker, the major bulk of work has to be done when executing the first query – as the data needs to be transformed to $1d$ space and a single query has to perform multiple cracking operations to avoid performance penalties due to the transformation to $1d$ space. Mosaic increases its cumulative time considerably due to its top-down partitioning strategy – it reorganizes data in frequently queried areas multiple times until it reaches its final level of refinement. Only the cumulative execution time of QUASII does not exceed the one of its static counter part, the R-Tree, in our experiments.

6.6.4 Comparative Analysis

In this set of experiments, we compare the performance of incremental approaches. We use the same setup as previously and measure the convergence of execution time as well as the cumulative execution time.

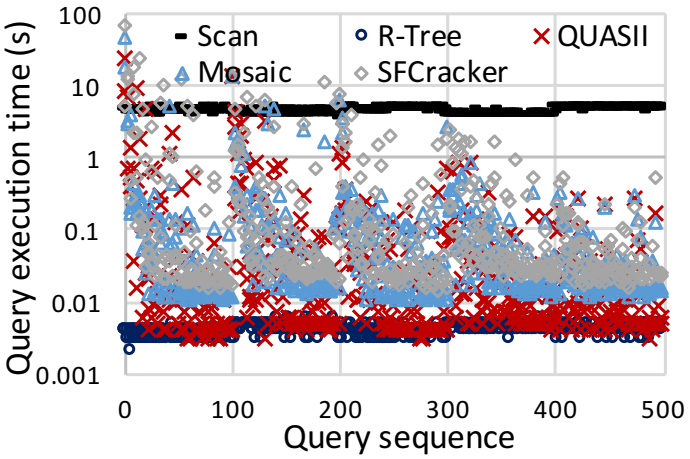


Figure 6.9 – Comparative analysis of incremental approaches: convergence.

Convergence. Figure 6.9 depicts the single query execution time for all the incremental approaches compared with the R-Tree and Scan. We use the R-Tree approach as a reference because it is the fastest approach among the static indexes for the workloads tested. We first analyze the execution time of the first query and then focus on the performance of the converged data structure.

The execution time of the first query determines data-to-insight time and thus has to be as small as possible. Among the incremental approaches, SFCracker has the most expensive first query due to the transformation of data to the $1d$ space. Mosaic's first query is faster, but still expensive as it has to reassign all the objects to new partitions, examining all three coordinates. Finally, QUASII has the least expensive first query due to the nested data reorganization – the number of objects necessary to be examined and reorganized becomes smaller as more dimensions are taken into account: all objects are scanned on the x-dimension, but on the y-dimension only the objects with a x-value satisfying the query will be scanned (accordingly for the z-dimension). Overall, Scan is 13.7, 9.2 and 4.6 times faster compared to SFCracker, Mosaic and QUASII respectively, when executing the first query.

Among the incremental approaches, only QUASII attains the query execution time of R-Tree on a fully converged index. Mosaic and SFCracker have at their core space-oriented partitioning and therefore, their performance is affected by the data assignment strategy as well as the skew in distribution, as Section 6.6.2 shows. SFCracker additionally transforms data to $1d$ domain and thus cannot preserve spatial proximity to the same extent as the other approaches. Consequently, QUASII outperforms Mosaic and SFCracker with a speedup of $3.68x$ and $4.9x$ respectively for the average execution time of a query in a fully refined area.

Cumulative Execution Time. We use the cumulative execution time as metric to evaluate the decrease in the data-to-insight time as well as the "break-even" point – the point when the cumulative cost of incremental exceeds that of static indexing – to assess the quality of an

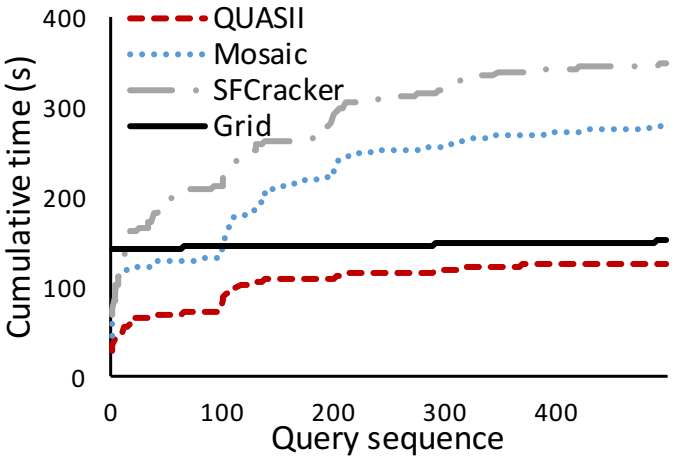


Figure 6.10 – Comparative analysis of incremental approaches: cumulative time.

incremental index. Figure 6.10 shows the experimental results. We use Grid as a reference since it has the smallest cumulative execution time among the static approaches – its pre-processing step is comparatively cheap (once its optimal configuration is determined).

As discussed in Section 6.6.3, SFCracker and Mosaic have comparatively expensive strategies and thus reach the performance of Grid after 13 and 100 queries respectively. Grid, on the other hand, compared to QUASI, has not amortized its building cost after 500 queries. More precisely, QUASII reaches 84% of the Grid cumulative execution time and, more importantly, it achieves 3.66x faster query performance for completely refined areas. QUASII executes the first query the fastest and consequently achieves the highest decrease in data-to-insight time – 5.1x and 11.4x compared to Grid and R-Tree.

For single query execution, the major benefit of QUASII comes from its data-oriented partitioning. Similar the to R-Tree, it adjusts to the distribution of the data and, as opposed to Grid and SFC, it does not replicate the objects or extend the query. It additionally keeps the data in multidimensional space and does consequently not suffer from decrease in dimensionality. Its low cumulative cost is mostly attributed to its incremental strategy. QUASII does not sort all objects, but rather reorganizes them within the specific bounds, gradually curbing the amount of data necessary to be reorganized.

Summary. QUASII outperforms other incremental approaches with respect to the convergence of execution time and cumulative time. It achieves performance comparable to the R-Tree (the fastest static approach) in the areas of the dataset where enough queries have been executed, while not exceeding the cumulative time of Grid (the static approach with the least expensive pre-processing). Its major benefits come from the data-oriented partitioning and the nested reorganization strategy which reorganizes precisely the data touched and used.

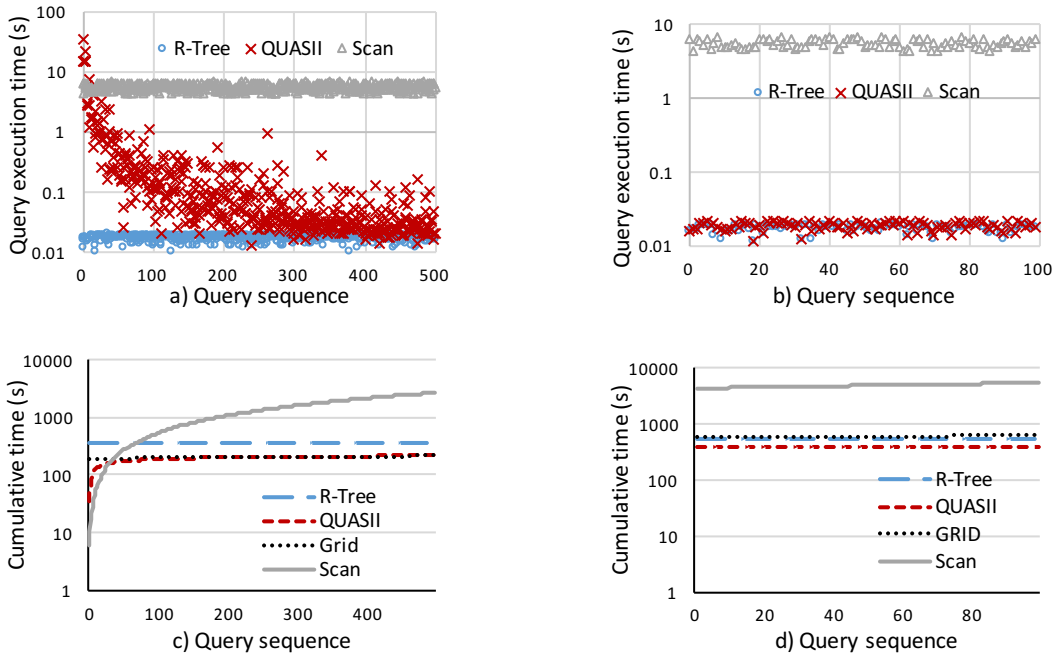


Figure 6.11 – Convergence and cumulative time: the first 500 (a & c) and last 100 (b & d) queries.

6.6.5 Analysis of QUASII

In this section we focus on QUASII. We evaluate its performance on the workloads other than neuroscience, analyze its scalability and the impact of query selectivity.

6.6.6 Uniform Workload

In the previous analyses we used workloads with query clusters that show the benefit of incremental approaches: the index quickly converges to the final performance as the queries are targeting the same areas. In this experiment we evaluate the performance of QUASII for a uniform workload. We execute 10000 uniformly distributed queries of selectivity 0.1% on the dataset with uniform distribution and 500M elements. We compare the performance of QUASII with Scan and R-Tree and additionally consider Grid for the cumulative execution time. Figure 6.11 illustrates both convergence and cumulative time for the first 500 and last 100 queries of the workload.

None of the first 500 queries is executed on a completely refined index. Starting with the 300th query, however, the single query execution is close to the final performance. Among the last 100 queries, 64 are executed on a completely refined index. The performance of queries on the refined structure is equal or very close to the performance of the R-Tree, i.e., on average 7.5% slower than the R-Tree.

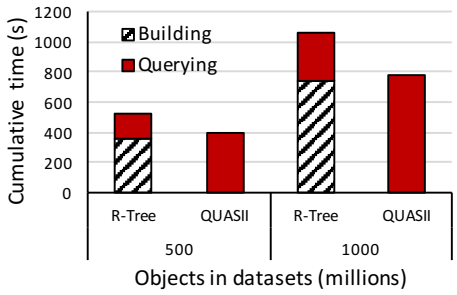


Figure 6.12 – Scalability analysis.

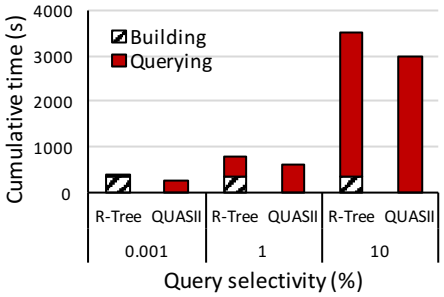


Figure 6.13 – Impact of selectivity.

After 10000 executed queries QUASII reaches 75% and 63.8% of the cumulative time of the R-Tree and Grid approaches respectively (y axis is in log scale). Likewise, it decreases data-to-insight time by 10.3x and 5.6x compared to R-Tree and Grid. Although the pre-processing step of Grid is significantly cheaper compared to the R-Tree, its cumulative time deteriorates with more queries executed due to the expensive single query performance.

6.6.7 Performance Trends

In the following experiment we evaluate the scalability of QUASII by executing 10000 queries of selectivity 0.1% on datasets with 500 million and 1 billion elements. In Figure 6.12 we compare the cumulative time of QUASII with R-Tree, where we additionally split the execution time of R-Tree into Building and Querying.

After 10000 executed queries QUASII reaches 75% and 73.7% of the cumulative time of the R-Tree with datasets of 500M and 1B elements respectively. By the time the R-Tree finishes its building process QUASII has executed around 8000 queries in both cases. QUASII also decreases data-to-insight time by 10.3x (on the 500M dataset) and 10.6x (on the 1B dataset) compared to the R-Tree, maintaining the performance trends as the dataset size increases.

6.6.8 Impact of Selectivity

In this set of experiments we evaluate the impact of query selectivity on the performance of QUASII. We measure the cumulative time for a uniform workload: 500M dataset and 5000 queries of 0.001%, 1%, and 10% selectivity. Figure 6.13 illustrates the results where we consider both the R-Tree and QUASII.

Intuitively, a static index (R-Tree) takes more time to amortize its building cost when executing 0.001% selectivity queries. On the other hand, the lower selectivity queries (10%) touch and reorganize a significant amount of data and QUASII thus reaches the break-even point with the R-Tree faster. Overall, after 5000 executed queries, QUASII reaches 68.8%, 79.8% and 85.6% of the cumulative time of the R-Tree for queries with 0.001%, 1%, and 10% selectivity.

6.7 Conclusions

The advances in data acquisition technologies and supercomputing for large-scale simulations rapidly increase the amounts of spatial data generated and collected. This data helps the scientist tremendously to gain insights and understand natural phenomena, however, at the same time, it leaves them with the challenge of analyzing it. Known approaches to spatial indexing have two major drawbacks with respect to exploratory analyses. First, they require a time-consuming pre-processing step that delays analyses. Second, given the massive amounts of data, a scientist frequently only analyzes a small fraction of it and consequently indexing the entirety of the data does not always pay off.

In this chapter we present a novel incremental index for the exploration of spatial data, where the ultimate goal is to let the scientists perform exploratory analyses as soon as the data is available, while using their queries to incrementally index the data. Our approach, QUASII, reduces data-to-insight time and curbs the cost of incremental indexing, by gradually and partially sorting the data, while producing a data-oriented hierarchical structure. As our experiments show, QUASII reduces the data-to-insight time by up to a factor of 11.4x, while its performance converges to that of the fastest state-of-the-art static indexes.

Part III

Dictionary Compression Tailored for Spatial Data

7 Dictionary Compression in Point Cloud Data Management

Nowadays, massive amounts of point cloud data can be collected thanks to advances in data acquisition and processing technologies like dense image matching and airborne LiDAR (Light Detection and Ranging) scanning. With the increase in volume and precision, point cloud data offers a useful source of information for natural resource management, urban planning, self-driving cars and more. At the same time, the scale at which point cloud data is produced, introduces management challenges: it is important to achieve efficiency both in terms of querying performance and space requirements. Traditional file-based solutions to point cloud management offer space efficiency, however, cannot scale to such massive data and provide the same declarative power as a database management system (DBMS).

In this chapter, we present a time- and space-efficient solution to storing and managing point cloud data in main memory column-store DBMS. Our solution, Space-Filling Curve Dictionary-Based Compression (SFC-DBC)[1], employs dictionary-based compression in the spatial data management domain and enhances it with indexing capabilities by using space-filling curves. It does so by constructing the space-filling curve over a compressed, artificially introduced 3D dictionary space. Consequently, SFC-DBC significantly optimizes query execution, and yet it does not require additional storage resources, compared to traditional dictionary-based compression. With respect to space-filling curve-based approaches, it minimizes storage footprint and increases resilience to skew. As a proof of concept, we develop and evaluate our approach as a research prototype in the context of SAP HANA. SFC-DBC outperforms other dictionary-based compression schemes by up to 61% in terms of space and up to 9.4x in terms of query performance.

7.1 Introduction

Recent advances in laser technology [137] and image processing [44] have evolved the importance of point cloud data and challenges considering its management. The ease of gathering 3D point cloud data, together with its public availability, have made it more attractive to users.

[1] This work originally appeared in [102].

During recent years, many datasets have been released as open data. These datasets offer a useful source of information for natural resource management, urban planning and more, by modeling point data through up to 26 properties such as x, y, and z coordinates, angle of scan, and color. One such prominent dataset is the second national height map of the Netherlands (AHN2) [88], which was acquired through airborne and terrestrial scanning and contains 640 billion points.

Given the massive amounts of point cloud data, it is important to achieve efficiency in terms of both querying performance and storage footprint. Traditional solutions to point cloud data management are file-based: points are stored in files in a predefined format and processed by application-specific algorithms. These solutions typically employ efficient compression schemes, but 1) face scalability problems with respect to the increasing number and size of files to process and 2) lack the declarative power of a DBMS [8, 134]. Therefore, research in this area has recently shifted towards DBMSs, as many of the data management challenges encountered with the increasing point cloud data size have already been addressed in DBMS solutions [134]. Recent work [8, 37, 67, 134] illustrates the potential of column-store DBMSs to meet point cloud management requirements, but focuses mostly on processing performance and ignores storage considerations.

This chapter presents a design for storing and managing point cloud data in the context of column-store DBMSs, that is driven both by time and space efficiency requirements. More specifically, we employ dictionary-based compression (DBC) – a compression method frequently used in main-memory column stores [30, 68] – in the spatial domain and enhance it with indexing capabilities, minimizing both space and time requirements. The resulting technique, Space-Filling Curve Dictionary-Based Compression (SFC-DBC), compresses point cloud data using DBC, leveraging the frequent repetition of the values for x, y, and z coordinates across point cloud entries; this property is particularly evident for data obtained through image matching processing as it inherits the grid-like structure of images. DBC significantly minimizes space requirements. However, it is agnostic to spatial data properties. To preserve and exploit spatial data properties and thus optimize further for query execution, we combine DBC with Space-Filling Curve (SFC) order to design a new compression scheme.

According to our compression scheme, a point cloud entry is represented through its position in an artificially introduced 3D dictionary space and indexed using a SFC order. As we illustrate in our experimental results, SFC-DBC does not require additional space resources, and yet significantly optimizes query execution, compared to traditional DBC. With respect to the traditional space-filling curve-based approaches, it minimizes storage footprint and increases resilience to skew.

In particular, our contributions are:

- We explore different solutions to store and manage point cloud data, having dictionary-based compression as a first-class citizen.

- We develop SFC-DBC, a novel encoding scheme that employs dictionary-based compression in the spatial domain, enhancing it with indexing capabilities to provide time and space efficiency properties.
- We develop and evaluate our approach as a research prototype in the context of SAP HANA [30]. SFC-DBC outperforms other dictionary-based compression schemes by up to 61% in terms of space and up to 9.4x in terms of query performance.

The remainder of the chapter is structured as follows. We provide the background of our work, i.e., discuss DBC and SFC order in the context of point cloud data management in Section 7.2. We introduce our approach in Section 7.3 and discuss experimental evaluation in Section 7.4. Finally, we draw conclusions in Section 7.5.

7.2 Background

Our proposed solution combines dictionary-based compression (DBC) and space-filling curve (SFC) order to efficiently store and manage point cloud data. Therefore, in this section we discuss the choice of DBC and SFC order, describe traditional approaches to these techniques and outline their shortcoming and challenges when it comes to point cloud data management.

7.2.1 Dictionary-based Compression

Two major technologies that are used for point cloud data acquisition are LiDAR [137] and dense image matching [44]. LiDAR is fundamentally a distance technology that uses an emitted laser pulse to determine an object's distance from a sensor, while image processing technology acquires point cloud data through dense image matching of multiple overlapping aerial images. With recent technological advancements, dense image matching has gained popularity as it offers the same capabilities as LiDAR, at a lower price and finer resolution [44]. Whether the point cloud data is obtained through LiDAR or image matching technology, the values for x, y, and z coordinates (not the points themselves) repeat across point cloud entries frequently. The data obtained through image matching processing by default has these properties as it inherits the grid-like structure of images, while LiDAR data obtains these characteristics as the result of typically employed post-processing steps (e.g., thinning-out of data) [110, 137]. We take advantage of these patterns in data distribution by employing DBC, a method frequently used in main-memory column stores [30, 68].

Dictionary-based Compression. DBC compresses a column by mapping its domain to a list of continuous integer values, i.e., replacing wide values in the attribute domain with smaller codes. Its simplest form consists of a dictionary and an index vector (*IV*) [107, 118]. The dictionary stores the sorted distinct values of the column domain, while the *IV* maps each point to its position in the dictionary. The IV can be further compressed [138].

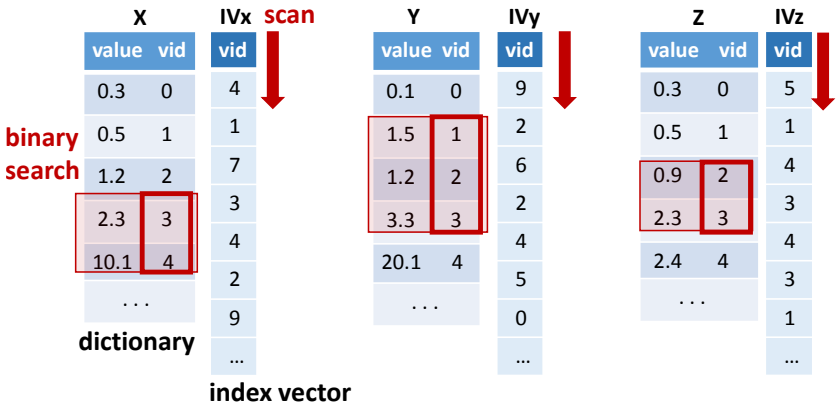


Figure 7.1 – An example of range query execution over a dictionary-based representation of point cloud data.

When naively applied in the context of point clouds, DBC represents point cloud data as three independent columns – one for each dimension of the 3D space – composed of a dictionary and an *IV*. The dictionary stores the sorted distinct values for the corresponding dimension and the *IV* maps the point to its corresponding position in the dictionary, as illustrated in Figure 7.1. A 3D range query is executed by performing binary search on the dictionary of each dimension to identify values and their corresponding positions in dictionaries that intersect with the query range. The binary search is followed by a scan of the corresponding *IV* to identify the records that match the identified dictionary position.

Challenges. We use the aforedescribed approach as a baseline in our experimental evaluation. For space-efficiency purpose, the *IV* is further compressed [138], i.e., each *IV* entry has a fixed length that corresponds to the number of bits necessary to represent a maximum value in *IV*.

Although the baseline DBC solution to store and query point cloud data is a straightforward one, it wastes computational resources as it does not leverage the spatial properties of data. It treats and consequently processes the dimensions and points independently, leading to a full scan of the *IVs* for each dimension. As we illustrate in our experiments (Section 7.4), even optimized scans of vector data require considerable time when processing massive point cloud datasets.

Therefore, to optimize this strategy we leverage a correlation within and across point cloud entries. A point cloud entry is represented with x, y, and z coordinates that are correlated, i.e., they describe a point in 3D space. Moreover, there is a correlation across the points: points close in 3D space will be frequently processed (e.g., queried) together. Therefore, we take advantage of this property and organize data to persevere spatial proximity. Consequently, we can restrict a search range using an index structure (that combines all 3 dimensions) and improve the data access patterns. However, a challenge is to achieve this in both time and space-efficient manner, as an index-like structure normally increases the storage footprint.

7.2.2 Space-filling Curves

A common way to preserve and exploit spatial data properties is by using Space-filling Curves (SFC) [69, 84, 85, 104, 134]. SFC-based organization transforms data from a multi- to a one-dimensional domain using a SFC to impose a total, one-dimensional (1D) order by visiting all the points in a D-dimensional grid exactly once. The Hilbert curve [59], the Gray-code curve [28], and the Z-order [96] are examples of SFC curves that are effective in preserving spatial proximity [29, 59, 86]. We opt to use a SFC-based organization to preserve and exploit spatial data properties due to its suitability for column-store DBMS. By transforming data to a 1D domain, we do not preserve spatial proximity to the same extent as with multi-dimensional data structures, however, we retain the ability to employ efficient scans of vector data. Simplicity and efficiency in the preprocessing step are additional benefits of this approach. In the following we describe the traditional approach to organize and query data using SFC. We focus on range queries as they are broadly used in many applications and are also the building block for many other spatial queries (e.g., k-nearest neighbor queries [60]).

SFC Organization. SFC order reorganizes data in three steps: (1) Partition the dataset's universe with a uniform grid and assign to each cell a value on the space-filling curve (*SFCcode*), (2) Assign *SFCcode* to every point cloud entry according to the grid cell they belong to, where multiple point cloud entries can map to the same *SFCcode* value, and (3) Sort the points based on the assigned *SFCcode*.

Range query execution is composed of two steps. 1) Transform a query to the 1D domain according to the SFC-order and perform binary search on the *SFCcodes* data structure based on the transformed ranges. 2) As a *SFCcode* is assigned per cell and not per point basis, all the points whose *SFCcode* matches the result of the binary search have to be additionally checked whether they belong to the query range in order to remove false positives. Techniques that partition the curve into multiple sub-intervals, each of which is fully contained in the original range [132], are used in order to minimize the number of checks in the second step.

Challenges. Traditional SFC-based organization offers a simple and efficient way to preserve and exploit spatial proximity. However, it does so by constructing a SFC order that is stored in addition to the data model. Therefore, whether we preserve data in the initial form (uncompressed 3D points) or use dictionary-based representation (Section 7.2.1), the *SFCcodes* structure requires additional storage resources. Consequently, applying the traditional scheme improves querying performance, however, it hurts space efficiency. To evaluate the performance of our approach, apart form the technique described in Section 7.2.1, we also develop a solution that is based on the aforedescribed, traditional approach. For space-efficiency purposes we introduce one modification – instead of representing data in its initial, uncompressed form, we use the dictionary-based representation (as discussed in the experimental evaluation section Section 7.4).

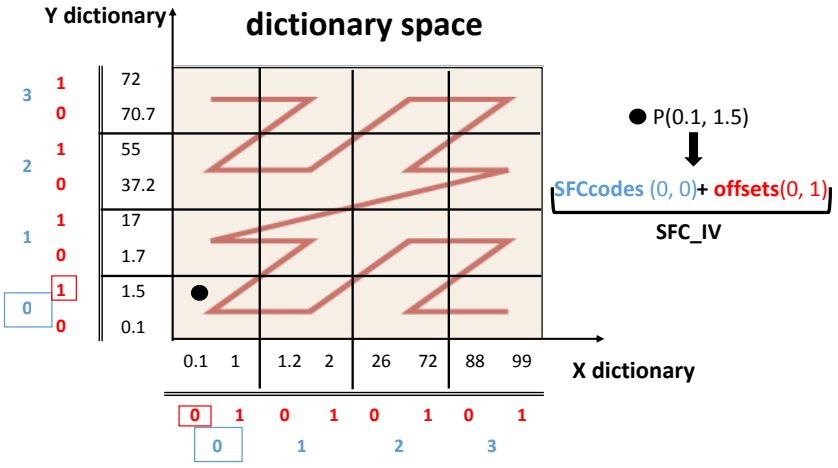


Figure 7.2 – Dictionary space, 2D illustration.

7.3 Space-Filling Curve Dictionary-Based Compression

To efficiently employ DBC in the spatial domain, we develop Space-Filling Curve Dictionary-Based Compression (SFC-DBC), a solution for storing and managing point cloud data that is driven both by time and space efficiency requirements. SFC-DBC combines DBC with SFC order to ensure space efficiency and preserve spatial proximity, thus optimizing for query execution. Our approach ultimately applies DBC in the spatial domain and enhances it with indexing capabilities without introducing additional storage requirements.

SFC-DBC represents a point cloud entry through its position in an artificially introduced 3D *dictionary space* and indexes it using a SFC order. The *dictionary space* is a compressed 3D space that we reconstruct from x, y, and z dictionaries. To do so, we exploit the fact that the dictionaries resemble the dataset space (universe) when combined, since each of them is sorted according to the corresponding dimension. SFC-DBC represents and indexes a point cloud entry using a SFC order constructed over this 3D space.

Figure 7.2 illustrates the *dictionary space* where, for the sake of simplicity, we use a 2D illustration. The SFC order (Z-order in our example) is constructed over the *dictionary space*, by partitioning it into four cells per dimension. SFC-DBC represents and indexes a point cloud entry with its position in the SFC order, i.e., with the assigned *SFCcode* which identifies the *dictionary space* cell that the point belongs to. As multiple points can map to the same *SFCcode*, to uniquely represent the position of the point in the *dictionary space* (and thus its value), SFC-DBC additionally captures the position within the cell that the point belongs to. For instance, in Figure 7.2 a point $P(0.1, 1.5)$ is represented through our encoding scheme with a *SFCcode* that encodes the cell ids that P maps to (0 and 0 value for x and y coordinate, indicated with blue color) and with the *offsets* that store the position of P within a cell (0 and 1 value for x and y coordinate, indicated with red color).

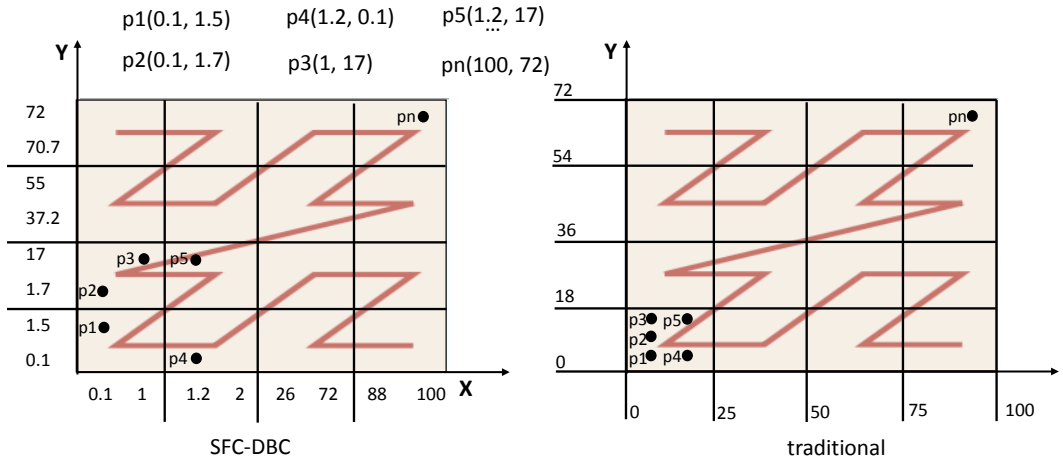


Figure 7.3 – SFC-DBC (data-aware) and SFC-based (space-oriented) partitioning strategy.

Improvement over DBC. With respect to traditional DBC, SFC-DBC significantly optimizes query execution, and yet it does not require additional storage resources, as we illustrate in Section 7.3.3. The key insight is that a SFC order is integrated into the dictionary space model. Consequently, the SFC order plays the role of the *IV* data structure (Section 7.2.1) while preserving spatial data proximity and low storage footprint.

Improvement over SFC. Compared to traditional SFC-based approaches, SFC-DBC minimizes storage footprint and increases resilience to skew. SFC-DBC achieves this by constructing a space-filling curve over a reduced *dictionary space*, instead of the original data space (universe). This partitioning strategy has a twofold effect on SFC-DBC. First, it enables the integration of SFC into the dictionary model. This consequently lowers the storage footprint and assigns two roles to the SFC: the role of spatial index and *IV* in DBC. Second, it enables a better adjustment of *SFCcodes* to the distribution of the data.

Figure 7.3 illustrates an example of data partitioning using both SFC-DBC and traditional SFC-based strategy. We use the subset of a dataset represented with six points p1-p5 & pn and assume that each dimension is divided into four cells. The SFC order, constructed according to the traditional encoding scheme, follows space-oriented partitioning, i.e., it uses uniform partitioning of the space, regardless of data distribution. As opposed to this, SFC order in SFC-DBC is defined in a data-aware manner. As illustrated in the example, data-aware partitioning improves skew handling, since it is done based on the actual points values. Data-aware also restricts the number of distinct points per cell, additionally improving skew resilience. In the example, SFC-DBC can have at most four distinct points per cell, while space-oriented partitioning does not have these guaranties.

It is necessary to notice that our partitioning scheme is a middle ground between traditional space- and data-oriented partitioning. The data-oriented strategy partitions data taking into consideration its spatial distribution. It also controls space utilization by limiting the number of objects assigned per partition. Our approach takes into consideration the data distribution,

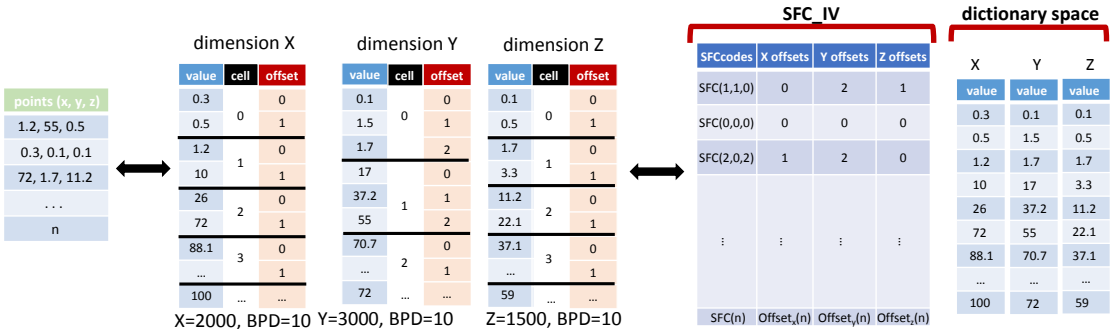


Figure 7.4 – Point cloud data organized according to SFC-DBC Encoding.

however it does not have explicit control over the number of elements assigned per partition (apart from an upper bound), given that it uses a grid-based structure.

In the following subsections we discuss necessary data structures and describe how to build and use them in the preprocessing and querying step. We discuss the space requirements of our approach and the impact of space-filling curve granularity on space- and time-efficiency of our approach. Finally, we conclude with the scope of our approach.

7.3.1 Preprocessing & Data Structures

The SFC-DBC approach represents a point cloud entry through its position in an artificially introduced 3D *dictionary space* and indexes it using a SFC order. Consequently, the preprocessing step results in two types of data structures: dictionary- and index-like structures. In the following we describe these structures and the preprocessing step that produces them.

Data Structures. SFC-DBC operates on a *dictionary space* and a *space-filling curve index vector* (*SFC_IV*) data structures, as illustrated in Figure 7.2 and Figure 7.4. The *dictionary space* is a $3D$ space reconstructed from x, y, and z dictionaries that captures the distinct values of point cloud entries in each dimension. *SFC_IV* maps a point to its position in the *dictionary space* and thereby its value. At the same time, it plays the role of a spatial index by encoding the point through its position in the SFC order constructed on top of the *dictionary space*.

To uniquely identify the position of a point in *dictionary space*, *SFC_IV* further consists of the *SFCcodes* and *offsets* vectors. The *SFCcodes* vector maps the point to its position in the SFC order based on the assigned *SFCcode*. The corresponding *SFCcode* does not uniquely identify the position of the point in the *dictionary space* but rather the cell it is in, as we assume that a point does not have a unique representation in the SFC order. Therefore, we also capture the position of the point within the cell using the *offsets* vector data structure.

We additionally compress both structures to further minimize the storage requirements. First, following the *IV* representation in the baseline DBC scheme (Section 7.2.1), we compress *offsets*

Algorithm 8: Query Execution: produce candidate results set

Input: q: range query - defined with two coordinates
Output: minDQ, maxDQ: min and max position in dictionary that corresponds to query range
Output: candidateSet: candidate result set

```
//transforms query to 1D:
for d = 0 to dimensions do
    minDQ[d] = binaryS(dictionary[d], q.low[d])
    maxDQ[d] = binaryS(dictionary[d], q.high[d])
end
qSFCcodes = calcSFCcode(q, minDQ, maxDQ)
candidateSet = binaryS(qSFCcodes, SFCcodes)
return candidateSet
```

vectors [138] – each entry has a fixed length that corresponds to the number of bits necessary to represent a maximum integer value in *offset*. Second, considering that multiple points map to the same *SFCcode*, we store just the distinct *SFCcodes* values and their corresponding starting positions in the input, similar to run-length encoding compression.

Preprocessing. The preprocessing step consists of four tasks that we describe through an example illustrated in Figure 7.4. First, we begin by producing a 3D *dictionary space* and a grid on top of it. More precisely, we produce a dictionary per dimension and divide them into cells, for each dimension independently. The number of cells is determined by the number of bits assigned per dimension (*BPD*) in the *SFCcode* and it corresponds to 2^{BPD}. For instance, in the example (Figure 7.4), the x dictionary is divided into 1024 cells as BPD corresponds to 10. Consequently, every cell has two dictionary entries, given that the number of entries in the x dictionary is 2000. Notice that the number of entries per cell (*EPC*) differs between x, y, and z dimensions as the dictionaries have different sizes (depending on the number of distinct values per dimension).

Second, once the *dictionary space* is partitioned, we assign a *SFCcode* to every point according to the dictionary cells they belong to. For instance, the first point $P(1.2, 55, 0.5)$ in the example (Figure 7.4, *points*) belongs to the cells with ids 1, 1 and 0 for the x, y, and z dimension respectively, and thus the *SFCcode* encodes these ids. The ids, however, do not uniquely identify the position of the point in the dictionaries. To do so, in the third step we additionally store the position of the point within the cells (Figure 7.4, *offsets*) – therefore, for the first point we store 0, 2 and 1 values according to x, y, and z dimensions. Lastly, once the final structures are produced, we sort them according to the assigned *SFCcode*.

7.3.2 Query Execution

Similar to the traditional SFC-based approach, the query execution is composed of two steps. In the first step, the *SFCcodes* vector restricts search space by producing the candidate results set, while in the second step we additionally prune, i.e., remove false positive results.

The first step is illustrated in Algorithm 8. The query execution first transforms the query range to the 1D domain, by determining its position in the *dictionary space* and calculating the corresponding *qSFCcodes*. Based on the produced codes, we determine the candidate result set by performing binary search on the *SFCcodes* vector. The resulting candidate set may contain false positive results considering that the *SFCcode* is assigned per cell and not per point. Therefore, in the second step we check if the identified points indeed belong to the query range.

To do so, we reconstruct the position of the point in the dictionary (and thus its value) for the points identified in the candidate results set and check if they belong to the query range. We perform this algorithm for all three dimensions in parallel, as illustrated in Algorithm 9. More precisely, the position is reconstructed by combining the decoded *SFCcode* and *offset* values, i.e., applying the following formula for the corresponding dimension:

$$position = cell\_id \times \#EPC + offsets \tag{7.1}$$

where $cell\_id$ represents the dictionary cell id obtained by decoding the *SFCcode* for the corresponding dimension and EPC stands for the number of entries per cell in the corresponding dictionary. Once the position is reconstructed, we check if it belongs to the query range $<minDQ, maxDQ>$, which corresponds to the minimum and maximum position in the dictionary that the query maps to (obtained in Algorithm 8).

Filtering. The time consuming operations in this process are the scan of the *offsets* vector and the decoding of the *SFCcode*. As the decoding is done once per distinct *SFCcode* value (once a value is decoded it is reused for all the points that have the same *SFCcode*), the scan of the *offsets* vector dominates the total execution time, as illustrated in Section 7.4.2. Therefore, to optimize the *offsets* vector scan, SFC-DBC minimizes the number of the offset entries necessary to be examined by skipping the entries that are completely enclosed by the query range. This can be done by checking the *enclosedByQuery* condition in Algorithm 9, which requires just the decoded *SFCcode* and EPC values in order to calculate the minimum and maximum position in dictionary that the points with a given *SFCcode* can map to. Intuitively, this optimization is more beneficial for the non-selective queries, as illustrated in Section 7.4.5.

7.3.3 Space Requirements

SFC-DBC enhances dictionary-based compression with indexing capabilities, optimizing for query execution without introducing additional space requirements. Therefore, in the following we analyze the space requirements of the baseline DBC and SFC-DBC.

Algorithm 9: Query Execution: produce final results set

Input: q: range query - defined with two coordinates
Input: candidateSet: candidate result set
Input: minDQ, maxDQ: min and max position in dictionary that corresponds to query range
Input: EPC: number of entries per cell, d - dimension
Output: pOut: point cloud result set

```
for i = value in candidateSet do
    cell_id = decode(SFCCodes[i], d)
    base = cell_id * EPC[d]
    //retrieve the positions of the points for the given SFCcode
    <inputMin, inputMax> = mapSFCcodeToInputPosition(i)
    //enclosedByQuery condition
    if minDQ[d] < base AND (base + EPC[d]) < maxDQ[d] then
        pOut.setRange(inputMin, inputMax)
        continue
    end
    //not enclosedByQuery - retrieve offsets
    for j = inputMin; j < inputMax do
        position = base + offests[j];
        if minDQ < position < maxDQ then
            pOut.set(j)
        end
    end
end
return pOut
```

As illustrated in Section 7.2.1, the baseline DBC operates based on *dictionaries* and *IV*. Therefore, its total space requirements correspond to Equation 7.2, where for the sake of simplicity we assume that each dictionary in 3D space has the same length. More precisely, $3 \times DS \times de$ represents the space requirements of the *dictionaries*, where DS and de are the number of entries and the size of an entry in the corresponding dictionary respectively. *IVs* corresponds to $3 \times n \times log_2 DS$, where n is the number of points and $log_2 DS$ is the number of bits per *IV* entry (which corresponds to the number of bits necessary to represent the maximum position in the corresponding dictionary).

$$3 \times (DS \times de + n \times log_2 DS) \tag{7.2}$$

On the other hand, the SFC-DBC approach stores *dictionaries*, *offsets* and *SFCcodes* vectors, resulting in the total space requirements illustrated in Equation 7.3. More precisely, the *dictionaries* are represented with $3 \times DS \times de$, the *offsets* vectors with $3 \times n \times log_2 \lceil DS/2^{BPD} \rceil$ where BPD is the number of bits assigned per dimension, while the *SFCcodes* vector corresponds to #SFCcode $\times 3 \times BPD$ where #SFCcode represents the number of distinct *SFCcode* values.

Compared to the space requirements of the baseline DBC – the *dictionaries* are identical, *offsets* vector minimizes the resources of *IV* since an offset entry indexes the values within a dictionary cell as opposed to the entire dictionary, while *SFCcodes* vector introduces additional space requirements.

$$\begin{aligned} & 3 \times (DS \times de + n \times log_2 \lceil DS/2^{BPD} \rceil + \#SFCcode \times BPD) \\ \equiv\ & 3 \times (DS \times de + n \times log_2 DS - n \times BPD + \#SFCcode \times BPD) \end{aligned} \tag{7.3}$$

Therefore, comparing the requirements of both approaches (according to Equation 7.2 and Equation 7.3), the SFC-DBC approach subtracts $3 \times n \times BPD$, while at the same time it introduces an additional overhead in the form of $3 \times \#SFCcode \times BPD$. Considering that $n \geqslant \#SFCcode$, the benefit is higher than the overhead and thus, the space requirements of SFC-DBC are always smaller or equal to the requirements of DBC.

7.3.4 Impact of Space-filling Curve

The granularity of the space-filling curve, i.e., the assigned number of bits per dimension (*BPD*) affects both time- and space-efficiency of our approach. Therefore, in the following we discuss in more detail and formalize this impact. Based on our analyses, we devise the models that can be used to determine the granularity of space-filling curve in order to optimize for time- and space-efficiency.

Time-efficiency. The *BPD* determines the total number of cells (per dimension) in the uniform grid built on top of the dictionary space and consequently the maximum number of distinct *SFCcodes*. The number of grid cells, i.e., the grid granularity controls the number of point cloud entries that qualify for the second step in the query execution (removing false positives). Therefore, by increasing *BPD* we boost query performance as we produce a finer-grained grid and reduce the number of points considered in the filtering step. However, by increasing *BPD* we also increase the number of distinct *SFCcodes* and consequently, the number of *SFCcodes* necessary to be decoded per query. Therefore, choosing a large number of *BPD* to represent the space-filling curve does not necessarily result in the best performance in terms of query execution. To further estimate the impact of the chosen granularity on the time-efficiency of SFC-DBC we devise a performance model. We concentrate on the second step in the query execution as it dominates the total execution time.

As described in Algorithm 9, given the identified *candidate result set*, i.e., the list of *SFCcodes* that overlap with the query range, SFC-DBC decodes the *SFCcodes* and checks if *SFCcodes*, i.e., the cells they belong to, are completely enclosed by the query range. The points that belong to the enclosed *SFCcode* are immediately marked as part of the results, while the points that belong to the border *SFCcodes* are further processed. More precisely, for a given *SFCcode* SFC-DBC reconstructs the position of the points in the dictionary using the corresponding offset values and checks if the positions belong to the query range.

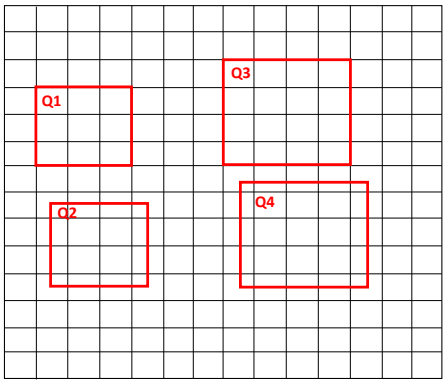


Figure 7.5 – The number of *SFCcodes* examined per query: the best and worst case.

Equation 7.4 models the cost of this process. $SFCcode_t$ is the total number of *SFCcodes* that are decoded and checked for enclosure, where $decode$ and $check_e$ represent the cost of aforementioned operations. $SFCcode_b$ is the number of border *SFCcodes* whose points are further checked for the intersection, where $check_i$ is the cost of the corresponding intersection checks. Finally, assuming uniform distribution, $n/\#SFCcode$ represents the number of points per a grid cell, where n is the total number of points in the dataset and *#SFCcode* is the maximum number of codes that can be generated given *BPD* and n, i.e., $\#SFCcode = min(2^{BPD \times d}, n)$ where d is the number of dimensions.

$$SFCcode_t \times (decode + check_e) + SFCcode_b \times \lceil \frac{n}{\#SFCcode} \rceil \times check_i \tag{7.4}$$

We first determine $SFCcode_t$ and $SFCcode_b$ – the number of total and border *SFCcodes* considered per query for a given *BPD* and the ratio query/universe size. Figure 7.5 illustrates an example of the worst and the best case scenario when a ratio between query and cell is 3 and 4 (note that we consider the ratio between corresponding edges). For the sake of simplicity we illustrate $2D$ example and assume a range query of equal edges.

The number of *SFCcodes* examined per query corresponds to the number of grid cells that intersect with the query. The best case minimizes the total number of intersecting cells and maximizes the number of completely enclosed cells. On the other hand, the worst case maximizes the total number of intersecting cells and minimizes the number of completely enclosed cells. In our example, queries *Q1* and *Q3* illustrate the best case (intersecting 9 and 16 cells), while queries *Q2* and *Q4* illustrate the worst case (intersecting 16 and 25 cells) for ratios 3 and 4, respectively. Equation 7.5 and Equation 7.6 generalize this rule and calculate the number of total and border cells (*SFCcodes*) for the best and worst case, where d stands for the number of dimensions and r for the query/cell edge ratio. In our model, we use the average of these extremes to express $SFCcode_t$ and $SFCcode_b$.

$$SFCcode_t = \lceil r \rceil^d, \quad SFCcode_b = \lceil r \rceil^d - \lfloor r \rfloor^d \tag{7.5}$$

$$SFCcode_t = (\lceil r \rceil + 1)^d, \quad SFCcode_b = (\lceil r \rceil + 1)^d - (\lfloor r \rfloor - 1)^d \tag{7.6}$$

Equation 7.5 and Equation 7.6 represent the best and the worst case for $SFCcode_t$ and $SFCcode_b$ independent of the dataset size. More precisely, the equations are devised given the assumption that all gird cells are represented with a *SFCcode*, i.e., have at least one point. However, this assumption does not hold if $n < 2^{BPD \times d}$, i.e., the number of points in a dataset is smaller than the maximal number of *SFCcodes* that can be generated given the *BPD* and *d*. Therefore, to take into consideration the actual dataset size, we adjust the worst and best case with the ratio $min(2^{BPD \times d}, n)/2^{BPD \times d}$, as illustrated in Equation 7.7 and Equation 7.8. $min(2^{BPD \times d}, n)$ is the maximal number of *SFCcodes* that can be produced given a dataset of size n, and $2^{BPD \times d}$ is the maximal number of *SFCcodes* that can be generated given the *BPD* and *d*.

$$SFCcode_t = \frac{min(2^{BPD \times d}, n)}{2^{BPD \times d}} \times \lceil r \rceil^d, \quad SFCcode_b = \frac{min(2^{BPD \times d}, n)}{2^{BPD \times d}} \times (\lceil r \rceil^d - \lfloor r \rfloor^d) \tag{7.7}$$

$$\begin{aligned} SFCcode_t &= \frac{min(2^{BPD \times d}, n)}{2^{BPD \times d}} \times (\lceil r \rceil + 1)^d \\ SFCcode_b &= \frac{min(2^{BPD \times d}, n)}{2^{BPD \times d}} \times ((\lceil r \rceil + 1)^d - (\lfloor r \rfloor - 1)^d) \end{aligned} \tag{7.8}$$

To model the cost of $decode$, $check_e$ and $check_i$ we use a simple execution model, and use the average cost of instruction using a methodology similar to that found in [21]. The cost of decoding is represented with Equation 7.9. More precisely, to decode a *SFCcode* value we first need to retrieve the *SFCcode* (incurring the cost of *MemoryTransfer*) and perform the corresponding decoding arithmetic operations (with the cost of *ArithmeticInstrDec*). However, given that we perform a binary search on *SFCcodes* in the first step of query execution, the corresponding *SFCcode* is transferred from cache (rather than memory) – resulting in the total cost of $CacheTransfer + ArithmeticInstrDec$. The cost of $check_e$ is modeled as Equation 7.10, i.e., with the corresponding arithmetic operations (*ArithmeticInstrEncCheck*) necessary to perform enclosure check, described in Algorithm 9.

$$\begin{aligned} decode &= MemoryTransfer + ArithmeticInstrDec \\ &\equiv CacheTransfer + ArithmeticInstrDec \end{aligned} \tag{7.9}$$

$$check_e = ArithmeticInstrEncCheck \tag{7.10}$$

The cost of $check_i$ is represented by Equation 7.11. More precisely, to check if a *SFCcode*'s points belong to a query range we need to retrieve the corresponding offsets from the *offsets* vector, reconstruct the positions of the points in the dictionary and perform the corresponding intersection checks. Therefore, the total cost corresponds to $MemoryTransfer +$

$ArithmeticInstrInter$. However, given that we access all the points for a *SFCcode* in sequence, we only pay the cost of *MemoryTransfer* for the first point, while other points have a cost of *CacheTransfer* (the memory access pattern is trivial for a prefetcher). The fetching of the first point is more expensive as we jump to the different parts of *offsets* vector for the different *SFCcodes* given that a query is partitioned into multiple sub-intervals (see Section 7.2.2).

$$\begin{aligned} check_i first &= MemoryTransfer + ArithmeticInstrInter \\ check_i &= CacheTransfer + ArithmeticInstrInter \end{aligned} \tag{7.11}$$

Incorporating the cost of $decode$, $check_e$ and $check_i$, Equation 7.12 represents a final model, where we benchmark the machine to determine the values for $MemoryTransfer$, $CacheTransfer$ and $ArithmeticInstr$. Given the number of *BPD*, n and the ratio of query to universe size, Equation 7.12 models the SFC-DBC performance trend in terms of query execution and can be used to maximize time-efficiency of SFC-DBC.

$$\begin{aligned} SFCcode_t \times (CacheTransfer + ArithmeticInstrDec + ArithmeticInstrEncCheck) \\ + SFCcode_b \times (MemoryTransfer + ArithmeticInstrInter) \\ + SFCcode_b \times \lceil \frac{n}{\#SFCcode} - 1 \rceil \times (CacheTransfer + ArithmeticInstrInter) \end{aligned} \tag{7.12}$$

Space-efficiency. The granularity of the space-filling curve, i.e., the assigned number of bits per dimension (*BPD*) affects also the space-efficiency of our approach. The *BPD* determines the size of the *SFCcode*, however, having fewer *BPD* does not necessarily imply smaller space requirements as *BPD* balances the space requirements of *SFCcodes* and *offsets* data structures. To further estimate the impact of chosen granularity on the space-efficiency of SFC-DBC, we devise a model that quantifies our storage requirements. We consider the data structures that are affected by the granularity of space-filling curve.

SFC-DBC stores the distinct *SFCcodes* values and their mapping, i.e., the corresponding starting positions in the input which map the points to the *SFCcode* values. Therefore, the total space requirements of the space-filling curve correspond to the space requirements of distinct *SFCcodes* ($SFCcodes$) and the corresponding mapping entries ($mapping$). Additionally, to be able to uniquely identify the position of the point in the dictionary space, SFC-DBC stores an offset for every point – resulting in the final cost illustrated in Equation 7.13.

$$SFCcodes + mapping + offsets \tag{7.13}$$

The total space requirements of $SFCcodes$ are further represented as Equation 7.14, where the size of a *SFCcode* is $3 \times BPD$. We adjust the number of distinct $SFCcodes$ to the dataset size applying logic similar to determining the best and worst case for the number of $SFCcodes_t$ and $SFCcodes_b$.

$$\begin{aligned} SFCcodes &= \#SFCcode \times 3 \times BPD \\ &\equiv \frac{min(2^{BPD \times d}, n)}{2^{BPD \times d}} \times min(2^{BPD \times d}, n) \times 3 \times BPD \end{aligned} \tag{7.14}$$

Similarly to the requirements of *SFCcodes*, the cost of $mapping$ depends on the number of entries necessary to be mapped (i.e., the number of distinct *SFCcodes*) where the size of each entry is $log_2 n$, as formulated in Equation 7.15.

$$\begin{aligned} mapping &= \#mapping \times log_2 n \\ &\equiv \frac{min(2^{BPD \times d}, n)}{2^{BPD \times d}} \times min(2^{BPD \times d}, n) \times log_2 n \end{aligned} \tag{7.15}$$

Finally, the total space requirements of $offsets$ correspond to Equation 7.16, where for the sake of simplicity we assume that each dictionary in 3D space has the same length, *DS*. Since an offset is a position of a point in a dictionary cell, the number of bits necessary to represent an offset corresponds to the number of bits necessary to represent the maximum position within a cell – $log_2 \lceil DS/2^{BPD} \rceil$. Finally, we store an offset for every point ($\times n$) and for all three dimensions ($\times 3$).

$$offsets = 3 \times n \times log_2 \lceil DS/2^{BPD} \rceil \tag{7.16}$$

Therefore, incorporating the cost of *SFCcodes*, *mapping* and *offsets* in Equation 7.13 gives us a model that, given the number of *BPD* and the number of elements of dataset n, estimates the space requirements of the SFC-DBC approach. Combined together with the time-efficiency model, it can be used to tune the performance of our approach considering both query execution and storage requirements.

7.3.5 Scope

As discussed, DBC is a good approach for point cloud data representation, because of the repetition of values for the x, y, and z coordinates. A limitation for such dictionary-based solutions is that this property cannot be guaranteed for raw (unprocessed) point cloud data obtained through LiDAR technology, as it results in unstructured form. However, LiDAR data typically obtains these characteristics as the result of post-processing steps (e.g., thinning-out of data) [110, 137].

Additionally, our solution is primarily designed for a static use case, which is aligned with the static nature of point cloud data. SFC-DBC inherits this property from space-filling curves, which are typically suitable for static environments. More precisely, an insertion/update to a space-filling curve might require re-computation of the $SFCcodes$ (if the universe space is modified) or reordering of the objects, so that they conform to the space-filling curve order.

Given that our approach inherits similar limitations, an insertion/update into our encoding scheme is a possible, yet costly operation, as the whole or a part of preprocessing might need to be redone. While this is a general constraint of our approach, it is necessarily to notice that this limitation does not hurt our scheme in the context of SAP HANA. As we discuss in section 7.4, HANA has segments of storage that do not need to provide cheap single-insert or update operations due to the Delta/Main concept.

7.4 Experimental Evaluation

In this section we first describe the experimental setup and methodology and then evaluate the performance of the proposed approaches over real-world datasets.

Hardware Configuration. We run experiments on a SuSE Linux Enterprise Server 12 SP1 machine equipped with 4 Intel Xeon CPU E7-4880v2 processors at 2.50GHz and 512GB of RAM. Each processor has 15 cores with private L1 (32KB) and L2 (256KB) caches, as well as 37,5MB of shared L3 cache.

SAP HANA. HANA is an in-memory database that offers the possibility to store data in either a row-oriented or a column-oriented fashion. It has a unique way of handling inserts and updates. More precisely, each column partition has two segments, a read-optimized Main segment and a write-optimized Delta segment. Updates and inserts are written to the Delta segment, while Main segments are created by an asynchronous background task. As this process has access to all the column fragment's data, it can make an optimal decision on the type of the Main segment (such as our SFC-DBC container) to be created and its properties. It is necessary to notice that SAP HANA is the system that we used as a proof of concept to develop and evaluate our approach, however, the proposed solution can be integrated in any other main memory column-store DBMS.

Implementation. All indexing techniques are implemented in C++ and compiled with `GCC 4.8.5`. The list below summarizes the implementations that we study experimentally.

Baseline is the baseline dictionary-based compression approach described in Section 7.2.1.

SFC-DBC represents our approach, introduced in Section 7.3. We use the Z-order as a SFC order, due to its simplicity and the huge body of work on its efficient range query algorithms (e.g., [11, 120, 132]). In our approach, a *zcode* encodes the cell ids (for x, y, and z dimension) that a point cloud entry maps to in a uniform grid built over the *dictionary space* and, therefore, we represent a *zcode* as an integer. The *BPD* in a *zcode* determines the total number of cells (per corresponding dimension) in the uniform grid and, consequently, the maximum number of point cloud entries that can qualify for the second step in the query execution of our approach, i.e., removing false positives. We represent *zcode* with 32 bits (10 bits per dimension — *BPD*) as a trade-off between memory resources and precision (number of false positives to be filtered).

SFC stands for the Space-Filling Curve-based approach, which we implemented as a middle ground solution between the Baseline approach and SFC-DBC. The SFC approach does not require decoding of *SFCcode*, however, it needs additional space for its storage. Therefore, we use SFC to evaluate SFC-DBC: the overhead introduced with its decoding step, but also the storage benefits.

More precisely, the SFC approach corresponds to the traditional approach described in Section 7.2.2 with one modification. To have a fair comparison in terms of space requirements, we build a SFC order as an addition to the DBC model. Therefore, the SFC approach extends the Baseline approach with a SFC order, using space-oriented partitioning of the dataset universe. Consequently, it stores the *SFCcodes* vector in addition to the structures used in the Baseline approach. Like in the SFC-DBC approach, we use the Z-order and compress the *SFCcodes*, each represented with 32 bits. The query execution is adjusted to the DBC model. We first execute a query on the *SFCcodes* producing a *candidate results set*, while in the second step we remove false positive results by examining the actual points. Similar to the Baseline approach, the points are examined by combining the information from *dictionaries* and *IVs*. However, the scan of *IV* is restricted, as we examine just the ranges detected by the *candidate results set*.

Datasets. We use two types of datasets, obtained using dense image matching and LiDAR technologies. "Senatsverwaltung für Wirtschaft, Technologie und Forschung" and "Europäischer Fonds für regionale Entwicklung (EFRE)" provided the datasets that are generated using dense image matching. Berlin aerial scan has regular point distribution with 100 points per square meter, while terrestrial castle scan represents irregular point distribution and a varying point density. We also use AHN2 dataset [88], obtained using LiDAR technology. To evaluate the scalability of approaches, we sample the datasets uniformly, increasing the dataset size from 125 million to 5 billion points.

Queries. We produce one hundred 2D and 3D range queries that follow uniform distribution. We vary selectivity by increasing the queries' volume: 0.01%, 0.1%, 1%, 10%, 20%, and 40%.

Experimental Results. In all the experiments we illustrate relative numbers. To preserve trends in the results, the numbers are represented relative to the smallest value in a result set (always having value 1 in the relative representation). For instance, the execution time is relative to the Baseline approach, i.e., its execution time obtained when processing the smallest dataset (125M elements). Following the same logic, the space requirements results are relative to the Uncompressed storage model (considering the smallest dataset).

7.4.1 Space Requirements

In this set of experiments we evaluate the space requirements of the aforementioned approaches when processing the Berlin aerial and AHN2 datasets. We measure the requirements of *dictionaries* and *IVs* for the Baseline approach, considering additionally the size of *SFCcodes* for the SFC-based approaches. Furthermore, we measure the size of uncompressed

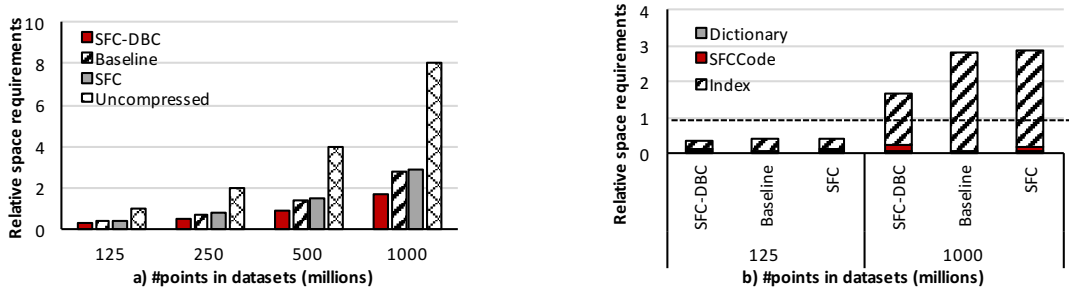


Figure 7.6 – Berlin aerial dataset, space requirements: a) total and b) breakdown.

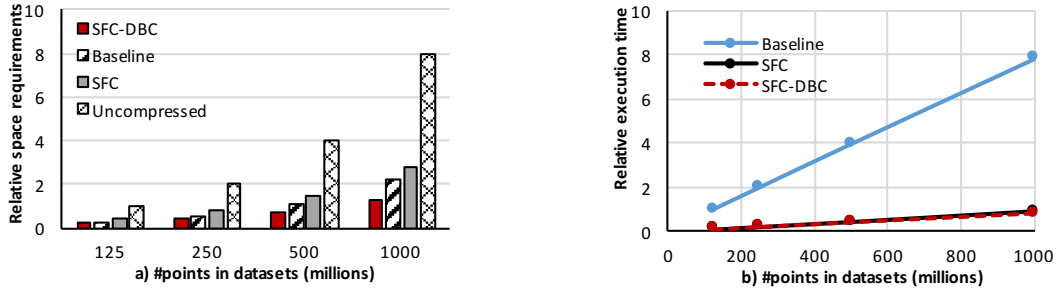


Figure 7.7 – AHN2 dataset: a) space requirements and b) query execution time (3D queries).

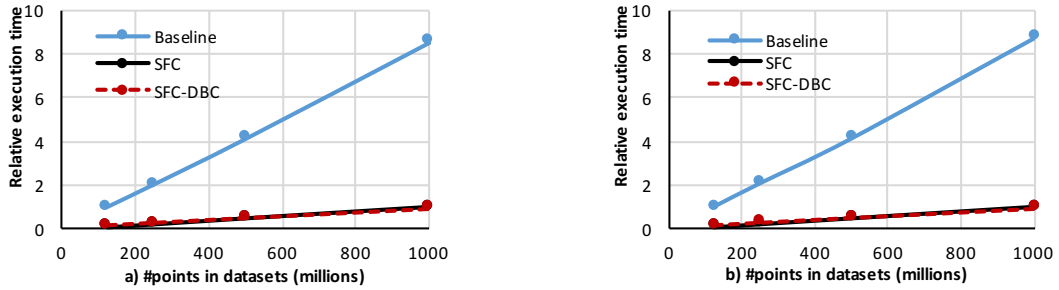


Figure 7.8 – Berlin aerial dataset, query execution time: a) 3D and b) 2D range queries.

data, i.e., when using a straightforward approach of storing all three coordinates. Figure 7.6a) presents the relative space requirements (Section 7.4 provides more details on relative values), while Figure 7.6b) illustrates the breakdown for the smallest and the biggest point cloud size when processing the Berlin aerial dataset. The horizontal line corresponds to the requirements of the Uncompressed model, for the smallest dataset.

Dictionary-based compression significantly reduces the space requirements: the Baseline approach reduces the space necessary to represent uncompressed data by up to 65%. SFC-DBC additionally minimizes the requirements, reducing the Baseline approach storage footprint by up to 40%. On the other hand, the SFC approach requires up to 13% more storage compared to the Baseline approach. The observed trends are similar for the AHN2 dataset, as illustrated in Figure 7.7a).

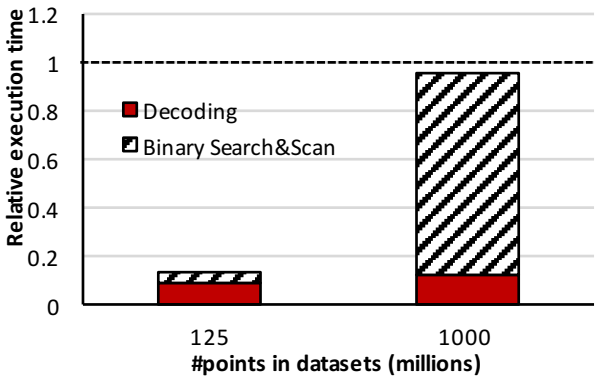


Figure 7.9 – SFC-DBC: execution time breakdown.

According to Figure 7.6b) all dictionary-based approaches have dictionaries of the same size, while the Index (i.e., *IV* / *offsets*) and the *SFCcodes* requirements vary. SFC-DBC has the smallest Index size, where the space reduction over the SFC and Baseline approaches is constant - 46%. On the other hand, it produces more distinct *SFCcodes* values compared to the SFC approach due to data-aware partitioning (as discussed in Section 7.3). Consequently, it increases the space quota, but it also improves skew handling by enabling a better adjustment of *SFCcodes* to the distribution of the data.

7.4.2 Query Performance

To evaluate the query performance of SFC-DBC we execute 100 uniformly distributed 3D and 2D range queries with a selectivity of 1% on the Berlin aerial and AHN2 datasets.

Figure 7.8 illustrates the relative execution time when executing 3D and 2D queries on the Berlin aerial dataset. As the experiment illustrates, the Baseline approach requires substantially more time because it scans the complete *IVs*, whereas the SFC-based approaches scan just the intervals detected by the candidate result set. Therefore, the SFC-DBC approach outperforms the Baseline approach with speedup of 7.5 - 8.9 and 6.8 - 9.4x, when executing 3D and 2D queries respectively. SFC-DBC achieves performance comparable to that of the SFC approach considering that 1) decoding is done once per distinct *SFCcode* value, and 2) the decoded value was sufficient to decide whether a point satisfies a range query for 86% of the candidate result set values (i.e., the filtering step was applied, Section 7.3.2). Therefore, SFC-DBC compensates for the decoding step by its ability to avoid the *offsets* vector scan. The observed trends are similar for the AHN2 dataset, as illustrated in Figure 7.7b).

Figure 7.9 presents the execution time breakdown for the SFC-DBC approach when processing the smallest and the biggest point cloud datasets. Decoding represents the time necessary to perform decoding of *SFCcode* and the Binary Search&Scan measures the time necessary to perform binary search (on *SFCcodes* and *dictionaries*) and the scan of the corresponding *offsets*. As the small dataset (125M points) is produced by uniformly sampling the biggest

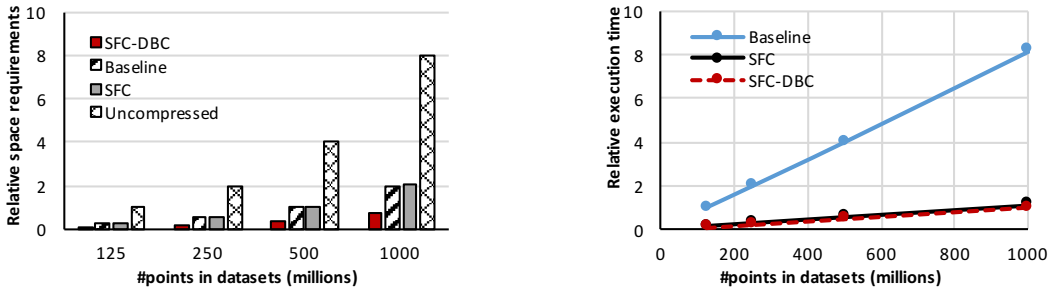


Figure 7.10 – The impact of skew: space requirements and query execution time.

dataset (1000M points), the number of distinct *SFCcode* values does not differ significantly between two datasets – the 1000M dataset has x1.32 more distinct *SFCcodes*. Therefore, the decoding time takes 71% and 13% of the total execution time, since the number of distinct *SFCcodes* corresponds to 29% and 5% of the point cloud entries in the smallest and biggest datasets respectively.

7.4.3 Impact of Data Skew

The performance of approaches based on space-oriented partitioning typically gets penalized when working with datasets that have non-uniform distributions. More precisely, space-oriented partitioning is done by partitioning the space containing the data, regardless of the spatial distribution of the objects. While space-oriented partitioning provides simplicity, it also limits the ability to adjust to the data distribution/density as it does not have explicit control over the number of elements assigned per partition.

Approaches based on traditional SFC partitioning inherit this problem, since they use space-oriented partitioning at their core. Given that our partitioning scheme is a middle ground between traditional space- and data-oriented partitioning, it also gets affected by skew in data. Therefore, in the following set of experiments we analyze the impact of skew in data on the space requirements and query performance of both SFC-based approaches. We execute 100 uniformly distributed 3D queries (1% selectivity) on the terrestrial castle scan that has an irregular point distribution and a varying point density. Figure 7.10a) presents space requirements, while Figure 7.10b) illustrates the execution time.

Due to skew in data distribution and density, the space requirements of the dictionary-based approaches are minimized compared to their requirements when processing data with uniform distribution (Section 7.4.1). As the number of distinct values per coordinate decreases, the space requirements of *dictionaries* and *IVs/offsets* also decrease and therefore, the Baseline approach reduces the space necessary to represent uncompressed data by up to 75%. Skew additionally reduces the number of distinct *SFCcodes* and thus, SFC-DBC reduces the Baseline approach storage footprint by up to 61%.

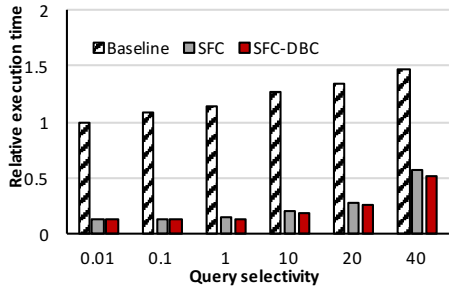


Figure 7.11 – The impact of selectivity.

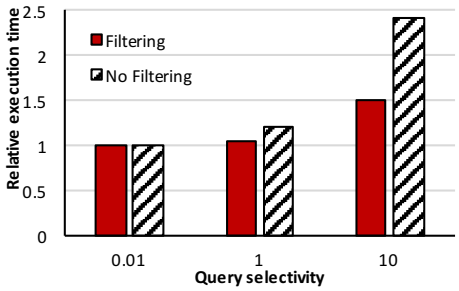


Figure 7.12 – The impact of filtering.

On the other hand, skew in data distribution and density reflects to *SFCcodes* distribution and thus penalizes query performance. More specifically, having fewer distinct *SFCcodes* increases the number of point cloud entries needed to be checked for intersection per *SFCcode*. Therefore, the speedup of SFC-DBC and SFC approaches over Baseline drops up to 10% and 16% compared to the speedup achieved when processing the data with uniform distribution.

The decrease in the number of distinct *SFCcodes* has a twofold effect on SCF-DBC performance. It hurts performance since the number of points necessary to be scanned increases. However, it also decreases the number of *SFCcodes* necessary to be decoded. Overall, SCF-DBC incurs smaller performance penalties, compared to the traditional scheme, considering that it employs data-aware partitioning and therefore, it adjusts better to data distribution, producing 1.59x more *SFCcodes*.

7.4.4 Impact of Selectivity

To evaluate the impact of selectivity on query performance we execute one hundred 3D queries on the Berlin aerial dataset (500 million points) varying selectivity from 0.01% to 40%. Figure 7.11 illustrates the total execution time.

As expected, SFC-based approaches benefit from queries with high selectivity. They minimize the number of *IV/offsets* entries necessary to be scanned using a SFC order as an index, while the Baseline approach performs a full *IV* scan. On the other hand, less selective queries (e.g., 40% selectivity) favour the Baseline approach considering that they touch significant amount of data. Thus, the speedup of the SFC-DBC approach drops from 8.4x to 2.8x when decreasing query selectivity.

7.4.5 Impact of Filtering

To evaluate the impact of filtering (introduced in Section 7.3.2) on the performance of SFC-DBC, we execute one hundred 3D queries on the Berlin aerial dataset (500 million points) varying selectivity from 0.01% to 10%. Figure 7.12 illustrates the relative execution time for the SFC-DBC approach, when we enable and disable filtering.

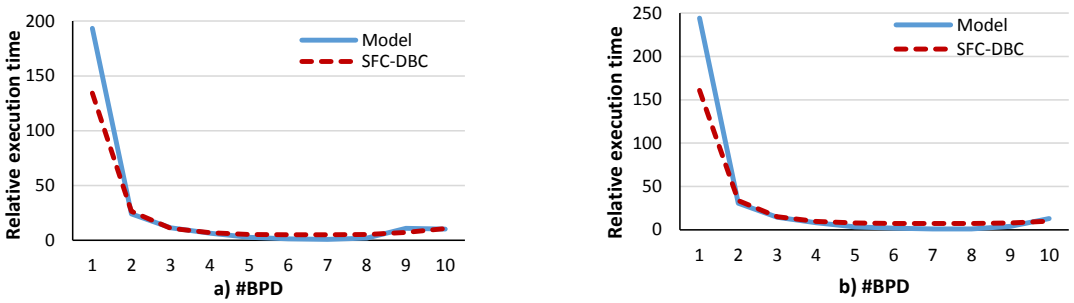


Figure 7.13 – The impact of space-filling curve: query execution time for a) 125M and b) 500M points.

The filtering step minimizes the range that has to be scanned in the *offsets* vector. Since the length of the range is determined by the query selectivity, the impact of filtering depends on the selectivity. As illustrated in the Figure 7.12, the filtering step significantly improves the execution time for low selectivity queries, considering that it filters more data (e.g., the improvement in the execution time is 38% for 10% selectivity). On the other hand, filtering does not have a significant impact on performance when executing high selectivity queries (e.g., for 0.01% selectivity queries the improvement in the execution time is 0.6%).

7.4.6 Impact of Space-filling Curve: Time-efficiency

In the following set of experiments we analyze the impact of the space-filling curve granularity on the performance of the SFC-DBC approach, when considering query execution time. At the same time, we evaluate the performance of the proposed performance model. We execute 100 uniformly distributed 3D queries (1% selectivity) on the AHN2 dataset of 125 and 500 million points. We vary the number of assigned bits per dimension *BPD* for the space-filling curve from 1 to 10 and measure the query execution time for each configuration. Figure 7.13 illustrates the experimental results where SFC-DBC represents the actual execution time and Model represents the execution time estimated based on our model.

We first analyze the measured execution time. Having a few *BPD* obviously hurts performance, as the space-filling curve has coarse granularity and consequently a significant percentage of the dataset has to be considered in the filtering step. Therefore, as we increase *BPD* we approach the best performance, which corresponds to the execution time when having 7 *BPD* (for both datasets). After that point, the total execution time gradually increases – resulting in 2.1x and 1.39x slower performance for 10 *BPD* when considering the 125M and 500M datasets, respectively. Therefore, the best configuration for this setting balances the number of points checked for intersection and the number of decoded *SFCcodes*. At this point, the space-filling curve achieves sufficient pruning and any additional refinement (i.e., increasing *BPD*) introduces overheads due to the additional decoding required.

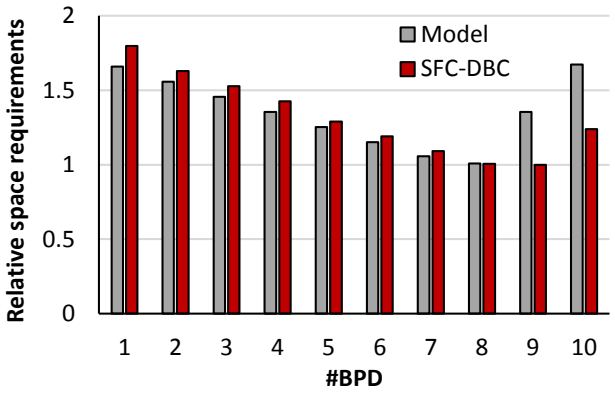


Figure 7.14 – The impact of space-filling curve on space-efficiency.

The execution time estimated based on our model achieves similar performance trends with the actual execution time. Model estimates 7 *BPD* as the best configuration for the dataset of 125M objects, which is aligned with the actual execution time. For the bigger dataset, of 500M elements, Model estimates 8 *BPD* as the best configuration. According to the actual execution time the 8 *BPD* configuration has 3% slower performance compared to the best configuration (7 *BPD*). It is necessary to notice that as we increse the number of elements in dataset, the configurations with the larger number of *BPD* become less penalized.

Model uses Equation 7.12 where we adjust the parameters given the benchmarks performed on our machine. The final model is illustrated in Equation 7.17, where U is the average cost of an integer instruction. $40 \times U$ represents the cost of fetching a *SFCcode* from L3 cache and $200 \times U$ is the cost of accessing the first *offset* for a *SFCcode* from memory. *ArithmeticInstrDec*, *ArithmeticInstr EncCheck*, *ArithmeticInstrInter* are represented with $2 \times BPD \times U$, $5 \times U$ and $4 \times U$ respectively.

$$\begin{aligned} & SFCcode_t \times (40 \times U + 2 \times BPD \times U + 5 \times U) \\ & + SFCcode_b \times (200 \times U + 4 \times U) \\ + SFCcode_b \times \lceil \frac{n}{\#SFCcode} - 1 \rceil & \times (20 \times U + 4 \times U) \end{aligned} \tag{7.17}$$

7.4.7 Impact of Space-filling Curve: Space-efficiency

To evaluate the impact of the space-filling curve granularity on storage requirements of the SFC-DBC approach and the accuracy of our space requirements model, we execute 100 uniformly distributed 3D queries (1% selectivity) on the AHN2 dataset of 500 million points. Similarly to the previous setting, we vary the number of assigned bits per dimension *BPD* for the space-filling curve from 1 to 10 and measure the storage requirements for each configuration. Figure 7.14 illustrates the results where SFC-DBC and Model represent the actual and estimated storage requirements.

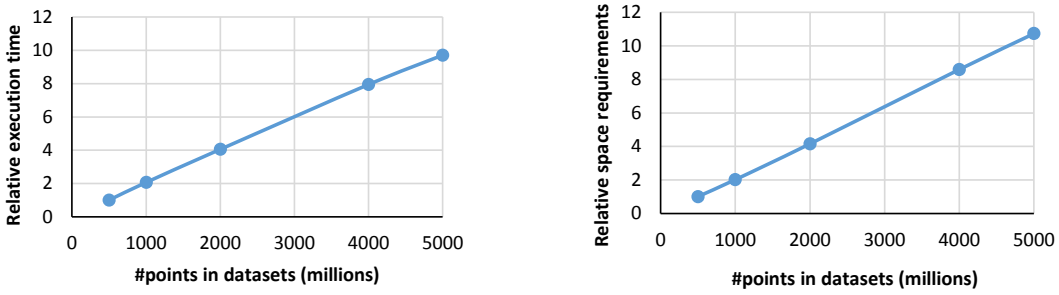


Figure 7.15 – SFC-DBC performance: a) query execution time and b) space requirements.

The best configuration, according to the actual storage requirements, corresponds to the configuration of 9 *BPD*. This configuration balances the space-requirements of *SFCcodes* and *offsets* data structures. A smaller number of bits in *BPD* results in smaller requirements in terms of *SFCcodes*, but higher in terms of *offsets*. On the other hand, the situation is opposite when increasing *BPD*.

According to our Model the best configuration corresponds to 8 *BPD*, while the actual storage requirements for 8 *BPD* take 1% more space compared to the best configuration (9 *BPD*). The minor difference in the estimation of the best configuration and the deviation in the performance trends for the configurations of 9 and 10 *BPD* are caused due to lack of knowledge about the actual data distribution. More precisely, the number of distinct *SFCcodes* for 9 and 10 *BPD* is significantly smaller compared to the estimated values.

7.4.8 Time- and Space-efficiency: Performance Trends

In this set of experiments we evaluate the scalability of our approach with respect to time and space requirements, when using the space-filling curve configuration suggested by our cost models. We use AHN2 datasets of 500, 1000, 2000, 4000, and 5000 million elements and execute 100 uniformly distributed 3D queries (1% selectivity).

The space-filling curve configuration (i.e., the number of assigned bits per dimension, *BPD*) that yields the best results in terms of time-efficiency does not necessarily align with the best configuration in terms of space-efficiency (as illustrated in Section 7.4.7 and Section 7.4.6). Therefore, based on our models, we calculate the best configuration in terms of time-efficiency (8, 9, 9, 9, 9) and space-efficiency (8, 8, 8, 9, 9) for each dataset (500, 1000, 2000, 4000, 5000 million elements) and use their average as the final configuration) to provide a middle ground between time- and space-efficiency. To evaluate the performance of our models, we determine the best configuration for our approach in terms of query performance and space utilization independently by varying *BPD* (as illustrated in Section 7.4.7 and Section 7.4.6).

Figure 7.15 presents the experimental results, i.e., query execution time and space requirements when using the BPD average configuration. As illustrated, SFC-DBC achieves scalability

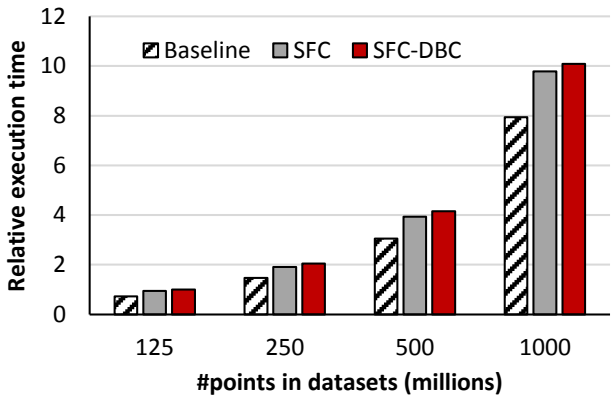


Figure 7.16 – Preprocessing cost.

both with respect to time and space requirements. Query execution time, however, scales slightly better, as the *BPD* configuration, used in our experiments, results in 1.5% slower execution time and uses 9.5% more storage requirements on average compared to the best query execution time and storage utilization identified by our benchmark.

7.4.9 Preprocessing cost

Our approach organizes data following the order established with a space-filling curve. This organization enables us to preserve and exploit spatial proximity, however, it also increases the preprocessing cost. To evaluate the overhead introduced with spatial reorganization we measure the total preprocessing time of the Baseline and SFC-based curve approaches when processing the Berlin aerial dataset. The experimental results are illustrated in Figure 7.16.

Compared to the Baseline approach, the SFC-based approaches additionally produce the *SFCcodes*, sort them and reorganize the data structures given this order. Consequently, the SFC-based approaches take up to 1.39x more time to preprocess the data compared to the Baseline approach. The performance of the two SFC-based approaches is comparable, where the minor difference in performance comes from the conceptual differences in the algorithms – SFC-DBC integrates the space-filling curve into the dictionary-based model, while the SFC approach produces it in addition to the model.

The data reorganization thus introduces an overhead into the preprocessing step, however, it also significantly boosts query performance given that it preserves spatial proximity.

7.4.10 Experimental Summary

The Baseline approach wastes computational resources as it requires a complete scan of index vectors. Adding a space-filling curve to the dictionary-based representation of point

cloud data, used in Baseline, restricts search space. However, the resulting approach requires additional storage resources, as illustrated in Section 7.4.1.

By integrating space-filling curves into the dictionary-based model, SFC-DBC does not require additional storage resources (Section 7.4.1), i.e., the space requirements of SFC-DBC are always smaller or equal to the requirements of the Baseline approach. SFC-DBC improves space savings over Baseline by adjusting the granularity of the space-filling curve. More precisely, having multiple points represented with the same $SFCcode$ enables us to additionally reduce the size of $SFCcode$ by using compression methods such as run-length encoding. In terms of time-efficiency, SFC-DBC preserves spatial proximity and reduces the search space by using the space-filling curve as an index (Section 7.4.2).

The granularity of the space-filling curve affects both space- and time-efficiency. In Section 7.4.7 and Section 7.4.6 we discuss and evaluate this impact, as well as the accuracy of our cost model devised for the space-filling configuration. Finally, our approach organizes data following the order established by a space-filling curve. While this reorganization enables us to preserve and exploit spatial proximity, it also increases the cost of preprocessing, as illustrated in Section 7.4.9.

7.5 Conclusions

With the recent increase in the volume of point cloud data produced, existing data management solutions face two challenges: time and space efficiency. In this work we investigate how the efficiency requirements can be met in main memory column-store DBMSs.

We propose Space-Filling Curve Dictionary-Based Compression (SFC-DBC), a time and space-efficient solution to storing and managing point cloud data. Our solution employs dictionary-based compression in the spatial data management domain, enhancing it with indexing capabilities without introducing additional storage overhead. The SFC-DBC approach represents and indexes a point cloud entry through its position in an artificially introduced 3D dictionary space, taking advantage of space-filling curve properties for indexing purposes. We evaluate our approach in the context of SAP HANA and show space efficiency gains of up to 61% and query performance gains of up to 9.4x compared to other dictionary-based compression schemes.

8 Conclusions and Future Directions

Spatial data analytics represent a powerful means to extract knowledge from data. However, due to recent technological advancements, there is a discrepancy between expectations, determined by application requirements, and the actual ability of spatial data management systems. In this thesis, we advocate for a data- and workload-aware design of spatial data management algorithms to bridge the gap between requirements and system performance.

In this chapter, we summarize the contributions of this dissertation and present potential directions for future work.

8.1 Technological Impact

Data-Aware Spatial Joins. The spatial join is a core operator in spatial analytics. Given its importance, a number of methods have been developed to perform spatial joins. However, as we show, the state-of-the-art spatial join approaches are unable to efficiently join two spatial datasets in a robust manner with respect to data distributions, as they employ static strategies in the join phase. Their performance deteriorates, or gets penalized with wasted computational resources when faced with variations in the distribution of data.

In Chapters 3 and 4 we argued that the key to optimizing performance is to be data-aware, that is to adapt the join strategy, and the supporting data structures to the underlying data distributions, in order to maximize performance. We introduced two disk-based spatial join approaches that leverage and adapt to dataset characteristics, achieving time-efficiency on non-uniform data distributions. Motivated by the increase in non-uniformity with respect to data distributions, we designed GIPSY to address the scenario of joining datasets with contrasting density. GIPSY uses the sparser dataset to navigate the join process and therefore, by leveraging dataset characteristics, it selectively retrieves and joins only the data needed. TRANSFORMERS achieves robust spatial joins on non-uniform data distributions, by adapting to dataset characteristics. It detects local variations in distributions and adapts the join strategy and data layout to local dataset characteristics at run-time.

Workload-Aware Spatial Incremental Indexing. Traditional systems require indexes to be built before analytical queries can be executed efficiently. Such an indexing step requires substantial computing resources and introduces a considerable and growing data-to-insight gap, where users need to wait before they can perform any analysis. Moreover, users often only use a small fraction of the data, the parts containing interesting information, and indexing it fully does not always pay off. This contrasts the latest data exploration trend propagated among data-driven applications, where the ultimate goal is to analyze data the moment is available, identify and extract useful information fast.

In Chapters 5 and 6 we advocated for workload-driven incremental index building, which significantly reduces the data-to-insight time and thus provides a tool for efficient data exploration. Indexes are built as a side-effect of query execution, and only for the parts of the data queried, consequently reducing computational and storage requirements. To our knowledge we are the first to develop and analyze incremental indexing for spatial data. We first introduced Space Odyssey, designed for exploratory analyses of multiple spatial datasets that reside on disk. Space Odyssey takes advantage of workload access patterns to incrementally index the datasets and optimize the access to parts frequently queried together. We then presented QUASII, a query-aware spatial incremental index. QUASII minimizes data-to-insight time, achieves efficient query performance and low-cost incremental strategy by preserving data in native, spatial domain and employing data-driven, nested reorganization strategy.

Dictionary Compression Tailored for Spatial Data. Point data representation and management are not new to spatial data management. However, with the recent advances in laser and image processing technology, data properties, and applications requirements in terms of point cloud management have evolved, challenging the efficiency of traditional solutions. More precisely, the scale at which point cloud data is produced demands for efficiency both in terms of querying performance and space requirements.

In Chapter 7 we argued that to maximize performance, apart from preserving spatial proximity, it is equally important to leverage secondary data characteristics, other than spatial. These characteristics are typically a result of technological advancements and introduce new patterns in data that should be exploited. We introduced the Space-filling curve dictionary-based compression approach that leverages point cloud data properties (the frequent repetition of values for the x, y, and z coordinates across point cloud entries), to employ dictionary-based compression in the spatial domain and enhance it with indexing capabilities to achieve both time- and space-efficiency. As a proof of concept, we developed and evaluated our approach as a research prototype in the context of SAP HANA.

8.2 Intellectual Impact

Our work on data-aware spatial joins demonstrates the potential of data-driven, adaptive strategies to maximize performance and avoid wasting computational resources. Given

varying data distributions, we identify strategies and data layouts that maximize performance with respect to the local data distributions and switch seamlessly between them.

The principles of our approaches are not tied to a disk-based environment or spatial joins approaches. Our solution minimizes the amount of data read and considered for intersection, and provides sequential access to data. This strategy can be beneficial in main memory as well, given that it minimizes computation and has the potential to reduce the number of cache misses. Looking beyond spatial joins, adapting the design of index structures to local data distributions has the potential to improve query performance and storage utilization. For instance, adjusting the granularity of partitions to the underlying data distributions, i.e., producing fewer partitions in densely populated areas, has the potential to minimize the problem of overlap associated with data-oriented partitioning.

Our work on incremental indexing sets the ground for techniques that enable efficient data exploration in the spatial domain. We illustrated the potential of workload-driven incremental index building to reduce the data-to-insight time and achieve query performance comparable to traditional indexing approaches. In addition, we explored the design space for incremental indexing in the spatial domain, by identifying promising design goals, but also by illustrating the limitations of applying existing one-dimensional solutions in the spatial domain.

Looking ahead, there are challenges to be addressed and opportunities that should be taken advantage of. One of the challenges is to achieve robustness with respect to workload access patterns, where the artificial refinement used in QUASII and the concepts from the one-dimensional domain [45] can be used as a base for more advanced techniques that can stabilize, but also optimize query performance. More precisely, a workload driven incremental strategy has the potential to not just minimize data-to-insight time, but also optimize query performance. By taking advantage of both workload access patterns and data properties, a data structure can be built incrementally, while adapting its form such that negative design implications are minimized – for instance, the overlap among partitions or the replication rate in data- and space-oriented partitioning strategies respectively.

Our work on point cloud data management enabled the use of dictionary-based compression in the spatial domain, while achieving both time- and space-efficiency. The employed scheme is thus a compact representation of point data, enhanced with indexing capabilities. In addition, it also represents a partitioning strategy that is a middle ground between data- and space-oriented partitioning, as it takes into consideration the data distribution, while preserving the simplicity of grid-like structures. The proposed scheme thus can be used to reduce storage requirements, but also as a partitioning strategy to improve query performance.

Bibliography

[1] *NYC Open Data*, 2018. http://data.ny.gov. [Page 2]

[2] Cristina L. Abad, Nathan Roberts, Yi Lu, and Roy H. Campbell. A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization, IISWC 2012, La Jolla, CA, USA, November 4-6, 2012*, pages 100–109, 2012. doi: 10.1109/IISWC.2012.6402909. URL https://doi.org/10.1109/IISWC.2012.6402909. [Page 86]

[3] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013. doi: 10.1561/1900000024. URL https://doi.org/10.1561/1900000024. [Page 3]

[4] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682, 2006. doi: 10.1145/1142473.1142548. URL http://doi.acm.org/10.1145/1142473.1142548. [Page 3]

[5] Ablimit Aji, Hoang Vo, and Fusheng Wang. Effective spatial data partitioning for scalable query processing. *CoRR*, abs/1509.00910, 2015. URL http://arxiv.org/abs/1509.00910. [Page 2]

[6] Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez, Omar Ghattas, Eui Joong Kim, Julio C. López, David R. O'Hallaron, Tiankai Tu, and John Urbanic. High resolution forward and inverse earthquake modeling on terascale computers. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, 15-21 November 2003, Phoenix, AZ, USA, CD-Rom*, page 52, 2003. doi: 10.1145/1048935.1050202. URL http://doi.acm.org/10.1145/1048935.1050202. [Page 2]

[7] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. Nodb: efficient query execution on raw data files. In *Proceedings of the ACM*

SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012, pages 241–252, 2012. doi: 10.1145/2213836.2213864. URL http://doi.acm.org/10.1145/2213836.2213864. [Pages 3, 4, and 16]

[8] Foteini Alvanaki, Romulo Goncalves, Milena Ivanova, Martin L. Kersten, and Kostis Kyzirakos. GIS navigation boosted by column stores. *PVLDB*, 8(12):1956–1959, 2015. doi: 10.14778/2824032.2824110. URL http://www.vldb.org/pvldb/vol8/p1956-alvanaki.pdf. [Pages 5, 18, and 114]

[9] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 570–581, 1998. URL http://www.vldb.org/conf/1998/p570.pdf. [Pages 12 and 16]

[10] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 347–358, 2004. doi: 10.1145/1007568.1007608. URL http://doi.acm.org/10.1145/1007568.1007608. [Pages 61 and 100]

[11] Rudolf Bayer. The universal b-tree for multidimensional indexing: general concepts. In *Worldwide Computing and Its Applications, International Conference, WWCA '97, Tsukuba, Japan, March 10-11, 1997, Proceedings*, pages 198–209, 1997. doi: 10.1007/3-540-63343-X_48. URL https://doi.org/10.1007/3-540-63343-X_48. [Pages 11, 14, 100, and 129]

[12] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990.*, pages 322–331, 1990. doi: 10.1145/93597.98741. URL http://doi.acm.org/10.1145/93597.98741. [Pages 12, 15, and 98]

[13] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. doi: 10.1145/361002.361007. URL http://doi.acm.org/10.1145/361002.361007. [Page 15]

[14] Jon Louis Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Software Eng.*, 5(4):333–340, 1979. doi: 10.1109/TSE.1979.234200. URL https://doi.org/10.1109/TSE.1979.234200. [Page 15]

[15] Renata Borovica-Gajic. Toward timely, predictable and cost-effective data analytics. 2016. [Page 3]

[16] Thomas Brinkhoff and Hans-Peter Kriegel. Approximations for a multi-step processing of spatial joins. In *IGIS '94: Geographic Information Systems, International Workshop on Advanced Information Systems, Monte Verita, Ascona, Switzerland, February 28 - March 4, 1994, Proceedings*, pages 25–34, 1994. doi: 10.1007/3-540-58795-0_31. URL https://doi.org/10.1007/3-540-58795-0_31. [Page 10]

[17] Thomas Brinkhoff, Hans-Peter Kriegel, and Ralf Schneider. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 40–49, 1993. doi: 10.1109/ICDE.1993.344079. URL https://doi.org/10.1109/ICDE.1993.344079. [Page 10]

[18] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993.*, pages 237–246, 1993. doi: 10.1145/170035.170075. URL http://doi.acm.org/10.1145/170035.170075. [Pages 12, 17, 23, 24, 27, 45, 46, and 48]

[19] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994.*, pages 197–208, 1994. doi: 10.1145/191839.191880. URL http://doi.acm.org/10.1145/191839.191880. [Page 10]

[20] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. Automatic contention detection and amelioration for data-intensive operations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 483–494, 2010. doi: 10.1145/1807167.1807221. URL http://doi.acm.org/10.1145/1807167.1807221. [Page 79]

[21] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 33–48, 2013. doi: 10.1145/2517349.2522714. URL http://doi.acm.org/10.1145/2517349.2522714. [Page 126]

[22] Jens-Peter Dittrich and Bernhard Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 535–546, 2000. doi: 10.1109/ICDE.2000.839452. URL https://doi.org/10.1109/ICDE.2000.839452. [Page 12]

[23] Harish Doraiswamy, Eleni Tzirita Zacharatou, Fábio Miranda, Marcos Lage, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. Interactive visual exploration of spatio-temporal urban data sets using urbane. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1693–1696, 2018. doi: 10.1145/3183713.3193559. URL http://doi.acm.org/10.1145/3183713.3193559. [Pages 2 and 3]

[24] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman, 2000. ISBN 978-0-8053-1755-8. [Pages 17, 24, 27, and 46]

[25] UBER Engineering. *Engineering Intelligence Through Data Visualization at Uber*, 2016. https://eng.uber.com/go-geofence/. [Page 1]

[26] UBER Engineering. *Uber Hits 5 Billion Rides Milestone*, 2017. https://www.uber.com/en-SG/blog/uber-hits-5-billion-rides-milestone/. [Page 1]

[27] UBER Engineering. *How We Built Uber Engineering's Highest Query per Second Service Using Go*, 2018. https://eng.uber.com/go-geofence/. [Page 3]

[28] Christos Faloutsos. Gray codes for partial match and range queries. *IEEE Trans. Software Eng.*, 14(10):1381–1393, 1988. doi: 10.1109/32.6184. URL https://doi.org/10.1109/32.6184. [Pages 11, 14, and 117]

[29] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, pages 247–252, 1989. doi: 10.1145/73721.73746. URL http://doi.acm.org/10.1145/73721.73746. [Pages 11, 88, 100, and 117]

[30] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012. URL http://sites.computer.org/debull/A12mar/hana.pdf. [Pages 114 and 115]

[31] Nivan Ferreira, Marcos Lage, Harish Doraiswamy, Huy T. Vo, Luc Wilson, Heidi Werner, Muchan Park, and Cláudio T. Silva. Urbane: A 3d framework to support data driven decision making in urban development. In *2015 IEEE Conference on Visual Analytics Science and Technology, VAST 2015, Chicago, IL, USA, October 25-30, 2015*, pages 97–104, 2015. doi: 10.1109/VAST.2015.7347636. URL https://doi.org/10.1109/VAST.2015.7347636. [Page 2]

[32] Georg Fuchs, Natalia V. Andrienko, Gennady L. Andrienko, Sebastian Bothe, and Hendrik Stange. Tracing the german centennial flood in the stream of tweets: first lessons learned. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Crowdsourced and Volunteered Geographic Information, GEOCROWD 2013, Orlando, FL, USA, November 5, 2013*, pages 31–38, 2013. doi: 10.1145/2534732.2534741. URL http://doi.acm.org/10.1145/2534732.2534741. [Page 2]

[33] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998. doi: 10.1145/280277.280279. URL https://doi.org/10.1145/280277.280279. [Pages 3, 4, 9, 10, 13, 14, 72, 73, 74, 86, and 87]

[34] Yván J. García, Mario A. López, and Scott T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *ACM-GIS '98, Proceedings of the 6th international symposium on Advances in Geographic Information Systems, November 6-7, 1998, Washington, DC, USA*, pages 163–164, 1998. doi: 10.1145/288692.288723. URL http://doi.acm.org/10.1145/288692.288723. [Pages 15, 61, and 100]

[35] Gartner. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*, 2018. https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016. [Page 1]

[36] S Gnanakaran, Hugh Nymeyer, John Portman, Kevin Y Sanbonmatsu, and Angel E Garcıa. Peptide folding simulations. *Current opinion in structural biology*, 13(2):168–174, 2003. [Page 1]

[37] Romulo Goncalves, Tom van Tilburg, Kostis Kyzirakos, Foteini Alvanaki, Panagiotis Koutsourakis, Ben van Werkhoven, and Willem Robert van Hage. A spatial column-store to triangulate the netherlands on the fly. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, pages 80:1–80:4, 2016. doi: 10.1145/2996913.2997005. URL http://doi.acm.org/10.1145/2996913.2997005. [Pages 5 and 114]

[38] Goetz Graefe and Harumi A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 371–381, 2010. doi: 10.1145/1739041.1739087. URL http://doi.acm.org/10.1145/1739041.1739087. [Pages 15 and 86]

[39] Goetz Graefe and Harumi A. Kuno. Adaptive indexing for relational keys. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 69–74, 2010. doi: 10.1109/ICDEW.2010.5452743. URL https://doi.org/10.1109/ICDEW.2010.5452743. [Page 15]

[40] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994.*, pages 243–252, 1994. doi: 10.1145/191839.191886. URL http://doi.acm.org/10.1145/191839.191886. [Page 79]

[41] Leopold Grinberg, T Anor, JR Madsen, A Yakhot, and GE Karniadakis. Large-scale simulation of the human arterial tree. *Clinical and Experimental Pharmacology and Physiology*, 36(2):194–205, 2009. [Pages 1 and 2]

[42] Ralf Hartmut Güting. An introduction to spatial database systems. *VLDB J.*, 3(4):357–399, 1994. URL http://www.vldb.org/journal/VLDBJ3/P357.pdf. [Pages 1, 3, and 9]

[43] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 47–57, 1984. doi: 10.1145/602259.602266. URL http://doi.acm.org/10.1145/602259.602266. [Pages 12, 14, 15, 17, 18, 27, and 48]

[44] Norbert Haala. Multiray photogrammetry and dense image matching. In *Photogrammetric Week*, volume 11, pages 185–195. Ed. D. Fritsch Heidelberg, 2011. [Pages 113 and 115]

[45] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012. doi: 10.14778/2168651.2168652. URL http://vldb.org/pvldb/vol5/p502_felixhalim_vldb2012.pdf. [Pages 4, 15, 88, and 143]

[46] Thomas Heinis, Farhan Tauheed, Mirjana Pavlovic, and Anastasia Ailamaki. Enabling scientific discovery via innovative spatial data management. *IEEE Data Eng. Bull.*, 36 (4):3–10, 2013. URL http://sites.computer.org/debull/A13dec/p3.pdf. [Pages 1, 2, and 3]

[47] Thomas Heinis, Farhan Tauheed, Mirjana Pavlovic, and Anastasia Ailamaki. Enabling scientific discovery via innovative spatial data management. *IEEE Data Eng. Bull.*, 36(4): 3–10, 2013. URL http://sites.computer.org/debull/A13dec/p3.pdf. [Page 3]

[48] Tony Hey, Stewart Tansley, and Kristin M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009. ISBN 978-0982544204. URL http://research.microsoft.com/en-us/collaboration/fourthparadigm/. [Page 1]

[49] Klaus H. Hinrichs. Implementation of the grid file: Design concepts and experience. *BIT*, 25(4):569–592, 1985. [Page 14]

[50] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. Twin grid files: Space optimizing access schemes. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988.*, pages 183–190, 1988. doi: 10.1145/50202.50222. URL http://doi.acm.org/10.1145/50202.50222. [Page 14]

[51] IDC. *The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things*, 2014. https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm. [Page 1]

[52] Stratos Idreos. Big data exploration. *Big Data Computing*, 2013. [Pages 3 and 4]

[53] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 68–78, 2007. URL http://cidrdb.org/cidr2007/papers/cidr07p07.pdf. [Pages 4, 15, 72, 86, 87, 88, 97, and 100]

[54] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011. doi: 10.14778/2002938.2002944. URL http://www.vldb.org/pvldb/vol4/p586-idreos.pdf. [Pages 4, 15, 72, and 88]

[55] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 277–281, 2015. doi: 10.1145/2723372.2731084. URL http://doi.acm.org/10.1145/2723372.2731084. [Pages 3 and 4]

[56] ANEC INLAS. *White paper big data*, 2016. https://portail-qualite.public.lu/dam-assets/fr/publications/normes-normalisation/information-sensibilisation/white-paper-big-data/WP_BigData_v1.pdf. [Page 1]

[57] Chris L Jackins and Steven L Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249–270, 1980. [Pages 11, 14, 18, and 89]

[58] Edwin H. Jacox and Hanan Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7, 2007. doi: 10.1145/1206049.1206056. URL http://doi.acm.org/10.1145/1206049.1206056. [Pages 4, 10, 16, 24, and 62]

[59] H. V. Jagadish. Linear clustering of objects with multiple atributes. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990.*, pages 332–342, 1990. doi: 10.1145/93597.98742. URL http://doi.acm.org/10.1145/93597.98742. [Pages 11, 14, 18, 31, 42, 88, and 117]

[60] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient b+-tree based indexing of moving objects. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 768–779, 2004. URL http://www.vldb.org/conf/2004/RS20P3.PDF. [Pages 88 and 117]

[61] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 500–509, 1994. URL http://www.vldb.org/conf/1994/P500.PDF. [Pages 15 and 100]

[62] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. Adaptive query processing on RAW data. *PVLDB*, 7(12):1119–1130, 2014. doi: 10.14778/2732977.2732986. URL http://www.vldb.org/pvldb/vol7/p1119-karpathiotakis.pdf. [Page 16]

[63] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015. URL http://cidrdb.org/cidr2015/Papers/CIDR15_Paper8.pdf. [Page 16]

[64] Kihong Kim, Sang Kyun Cha, and Keunjoo Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 139–150, 2001. doi: 10.1145/375663.375679. URL http://doi.acm.org/10.1145/375663.375679. [Page 15]

[65] Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 324–335, 1997. doi: 10.1145/253260.253340. URL http://doi.acm.org/10.1145/253260.253340. [Pages 11, 12, and 16]

[66] James Kozloski, Konstantinos Sfyrakis, Sean L. Hill, Felix Schürmann, Charles C. Peck, and Henry Markram. Identifying, tabulating, and analyzing contacts between branched neuron morphologies. *IBM Journal of Research and Development*, 52(1-2):43–56, 2008. doi: 10.1147/rd.521.0043. URL https://doi.org/10.1147/rd.521.0043. [Pages 25 and 49]

[67] Kostis Kyzirakos, Foteini Alvanaki, and Martin L. Kersten. In memory processing of massive point clouds for multi-core systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 7:1–7:10, 2016. doi: 10.1145/2933349.2933356. URL http://doi.acm.org/10.1145/2933349.2933356. [Pages 18 and 114]

[68] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. SQL server column store indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1177–1184, 2011. doi: 10.1145/1989323.1989448. URL http://doi.acm.org/10.1145/1989323.1989448. [Pages 114 and 115]

[69] Robert Laurini. Graphics databases built on peano space-filling curves. In *Proceedings of EUROGRAPHICS*, volume 85, pages 327–338, 1985. [Page 117]

[70] Scott T. Leutenegger, J. M. Edgington, and Mario A. López. STR: A simple and efficient algorithm for r-tree packing. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, pages 497–506, 1997. doi: 10.1109/ICDE.1997.582015. URL https://doi.org/10.1109/ICDE.1997.582015. [Pages 15, 29, 51, 61, 86, 92, and 100]

[71] Yanhui Liang, Fusheng Wang, Darren Treanor, Derek R. Magee, George Teodoro, Yangyang Zhu, and Jun Kong. Liver whole slide image analysis for 3d vessel reconstruction. In *12th IEEE International Symposium on Biomedical Imaging, ISBI 2015, Brooklyn, NY, USA, April 16-19, 2015*, pages 182–185, 2015. doi: 10.1109/ISBI.2015.7163845. URL https://doi.org/10.1109/ISBI.2015.7163845. [Page 2]

[72] Yanhui Liang, Fusheng Wang, Darren Treanor, Derek R. Magee, George Teodoro, Yangyang Zhu, and Jun Kong. A 3d primary vessel reconstruction framework with serial microscopy images. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*, pages 251–259, 2015. doi: 10.1007/978-3-319-24574-4_30. URL https://doi.org/10.1007/978-3-319-24574-4_30. [Page 2]

[73] Yanhui Liang, Hoang Vo, Ablimit Aji, Jun Kong, and Fusheng Wang. Scalable 3d spatial queries for analytical pathology imaging with mapreduce. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, pages 52:1–52:4, 2016. doi: 10.1145/2996913.2996925. URL http://doi.acm.org/10.1145/2996913.2996925. [Page 2]

[74] Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994.*, pages 209–220, 1994. doi: 10.1145/191839.191881. URL http://doi.acm.org/10.1145/191839.191881. [Pages 12 and 17]

[75] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 247–258, 1996. doi: 10.1145/233269.233337. URL http://doi.acm.org/10.1145/233269.233337. [Page 17]

[76] Nikos Mamoulis. Spatial join. In *Encyclopedia of Database Systems*, pages 2707–2714. 2009. doi: 10.1007/978-0-387-39940-9_356. URL https://doi.org/10.1007/978-0-387-39940-9_356. [Pages 4 and 16]

[77] Nikos Mamoulis. *Spatial Data Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011. doi: 10.2200/S00394ED1V01Y201111DTM021. URL https://doi.org/10.2200/S00394ED1V01Y201111DTM021. [Pages 1, 3, 9, and 13]

[78] Nikos Mamoulis and Dimitris Papadias. Multiway spatial joins. *ACM Trans. Database Syst.*, 26(4):424–475, 2001. doi: 10.1145/503099.503101. URL http://doi.acm.org/10.1145/503099.503101. [Page 4]

[79] Nikos Mamoulis and Dimitris Papadias. Slot index spatial join. *IEEE Trans. Knowl. Data Eng.*, 15(1):211–231, 2003. doi: 10.1109/TKDE.2003.1161591. URL https://doi.org/10.1109/TKDE.2003.1161591. [Page 17]

[80] Nikos Mamoulis, Yannis Theodoridis, and Dimitris Papadias. Spatial joins: Algorithms, cost models and optimization techniques. In *Spatial Databases: Technologies, Techniques and Trends*, pages 155–184. 2005. [Page 16]

[81] Henry Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153, 2006. [Page 24]

[82] Henry Markram, Karlheinz Meier, Thomas Lippert, Sten Grillner, Richard S. Frackowiak, Stanislas Dehaene, Alois Knoll, Haim Sompolinsky, Kris Verstreken, Javier DeFelipe, Seth Grant, Jean-Pierre Changeux, and Alois Saria. Introducing the human brain project. In *Proceedings of the 2nd European Future Technologies Conference and Exhibition, FET 2011, Budapest, Hungary, May 4-6, 2011*, pages 39–42, 2011. doi: 10.1016/j.procs.2011.12.015. URL https://doi.org/10.1016/j.procs.2011.12.015. [Pages 1, 2, 45, 49, 71, 78, 85, and 87]

[83] Henry Markram, Eilif Muller, Srikanth Ramaswamy, Michael W Reimann, Marwan Abdellah, Carlos Aguado Sanchez, Anastasia Ailamaki, Lidia Alonso-Nanclares, Nicolas Antille, Selim Arsever, et al. Reconstruction and simulation of neocortical microcircuitry. *Cell*, 163(2):456–492, 2015. [Page 87]

[84] Oscar Martinez-Rubi, Peter van Oosterom, Romulo Goncalves, Theo Tijssen, Milena Ivanova, Martin L. Kersten, and Foteini Alvanaki. Benchmarking and improving point cloud data management in monetdb. *SIGSPATIAL Special*, 6(2):11–18, 2014. doi: 10.1145/2744700.2744702. URL http://doi.acm.org/10.1145/2744700.2744702. [Pages 18, 19, and 117]

[85] Mohamed F. Mokbel and Walid G. Aref. Space-filling curves for query processing. In *Encyclopedia of Database Systems*, pages 2675–2680. 2009. doi: 10.1007/978-0-387-39940-9_350. URL https://doi.org/10.1007/978-0-387-39940-9_350. [Page 117]

[86] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Trans. Knowl. Data Eng.*, 13(1):124–141, 2001. doi: 10.1109/69.908985. URL https://doi.org/10.1109/69.908985. [Pages 11, 88, and 117]

[87] NASA. *NASA/NEX Public Data Set*, 2017. https://aws.amazon.com/blogs/aws/process-earth-science-data-on-aws-with-nasa-nex/. [Pages 2 and 85]

[88] Actueel Hoogte Bestand Nederland. *AHN datasets*, 2017. http://www.ahn.nl. [Pages 2, 85, 114, and 130]

[89] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984. doi: 10.1145/348.318586. URL http://doi.acm.org/10.1145/348.318586. [Pages 12 and 14]

[90] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 701–712, 2013. doi: 10.1145/2463676.2463700. URL http://doi.acm.org/10.1145/2463676.2463700. [Pages 4 and 23]

[91] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *PVLDB*, 10(10):1106–1117, 2017. doi: 10.14778/3115404.3115415. URL http://www.vldb.org/pvldb/vol10/p1106-olma.pdf. [Page 16]

[92] Matthaios Olma, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. BLOCK: efficient execution of spatial range queries in main-memory. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*, pages 15:1–15:12, 2017. doi: 10.1145/3085504.3085519. URL http://doi.acm.org/10.1145/3085504.3085519. [Page 14]

[93] Beng Chin Ooi. Spatial kd-tree: A data structure for geographic database. In *Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Darmstadt, 1.-3. April 1987, Proceedings*, pages 247–258, 1987. doi: 10.1007/978-3-642-72617-0_17. URL https://doi.org/10.1007/978-3-642-72617-0_17. [Page 15]

[94] OpenStreetMap. https://www.openstreetmap.org. URL https://www.openstreetmap.org. [Pages 2 and 85]

[95] Oracle. *Spatial and Graph Developer's Guide*, 2017. https://docs.oracle.com/database/121/SPATL/. [Page 18]

[96] Jack A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 181–190, 1984. doi: 10.1145/588011.588037. URL http://doi.acm.org/10.1145/588011.588037. [Pages 11, 14, 18, 88, and 117]

[97] Stratos Papadomanolakis, Anastassia Ailamaki, Julio C. López, Tiankai Tu, David R. O'Hallaron, and Gerd Heber. Efficient query processing on unstructured tetrahedral meshes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 551–562, 2006. doi: 10.1145/

1142473.1142535. URL http://doi.acm.org/10.1145/1142473.1142535. [Pages 24, 28, 29, and 50]

[98] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 259–270, 1996. doi: 10.1145/233269.233338. URL http://doi.acm.org/10.1145/233269.233338. [Pages 11, 12, 16, 23, 24, 27, 45, 46, and 48]

[99] Mirjana Pavlovic, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. GIPSY: joining spatial datasets with contrasting density. In *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, pages 11:1–11:12, 2013. doi: 10.1145/2484838.2484855. URL http://doi.acm.org/10.1145/2484838.2484855. [Pages 7, 23, 46, and 48]

[100] Mirjana Pavlovic, Thomas Heinis, Farhan Tauheed, Panagiotis Karras, and Anastasia Ailamaki. TRANSFORMERS: robust spatial joins on non-uniform data distributions. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 673–684, 2016. doi: 10.1109/ICDE.2016.7498280. URL https://doi.org/10.1109/ICDE.2016.7498280. [Pages 7 and 45]

[101] Mirjana Pavlovic, Eleni Tzirita Zacharatou, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. Space odyssey: efficient exploration of scientific data. In *Proceedings of the Third International Workshop on Exploratory Search in Databases and the Web, San Francisco, CA, USA, July 1, 2016*, pages 12–18, 2016. doi: 10.1145/2948674.2948677. URL http://doi.acm.org/10.1145/2948674.2948677. [Pages 7, 71, 86, and 87]

[102] Mirjana Pavlovic, Kai-Niklas Bastian, Hinnerk Gildhoff, and Anastasia Ailamaki. Dictionary compression in point cloud data management. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, pages 45:1–45:10, 2017. doi: 10.1145/3139958.3139969. URL http://doi.acm.org/10.1145/3139958.3139969. [Pages 7 and 113]

[103] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. QUASII: query-aware spatial incremental index. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, pages 325–336, 2018. doi: 10.5441/002/edbt.2018.29. URL https://doi.org/10.5441/002/edbt.2018.29. [Pages 7 and 85]

[104] Giuseppe Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, 1890. [Page 117]

[105] Nikos Pelekis and Yannis Theodoridis. *Mobility Data Management and Exploration*. Springer, 2014. ISBN 978-1-4939-0391-7. doi: 10.1007/978-1-4939-0392-4. URL https://doi.org/10.1007/978-1-4939-0392-4. [Pages 1, 3, and 9]

[106] PostgreSQL. *A PostgreSQL extension for storing point cloud (LiDAR) data*, 2017. https://github.com/pgpointcloud/pointcloud. [Page 18]

[107] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling up concurrent main-memory column-store scans: Towards adaptive numa-aware data and task placement. *PVLDB*, 8(12):1442–1453, 2015. doi: 10.14778/2824032.2824043. URL http://www.vldb.org/pvldb/vol8/p1442-psaroudakis.pdf. [Page 115]

[108] rapidlasso GmbH. *LAStools*, 2017. https://rapidlasso.com/lastools/. [Page 17]

[109] Gautam Ray, Jayant R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, page 0, 1995. [Page 3]

[110] Rico Richter and Jürgen Döllner. Out-of-core real-time visualization of massive 3d point clouds. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, Afrigraph 2010, Franschhoek, South Africa, June 21-23, 2010*, pages 121–128, 2010. doi: 10.1145/1811158.1811178. URL http://doi.acm.org/10.1145/1811158.1811178. [Pages 115 and 128]

[111] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984. doi: 10.1145/356924.356930. URL http://doi.acm.org/10.1145/356924.356930. [Pages 11, 14, and 18]

[112] Hanan Samet and Walid G. Aref. Spatial data models and query processing. In *Modern Database Systems*, pages 338–360. 1995. [Pages 3 and 9]

[113] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 507–518, 1987. URL http://www.vldb.org/conf/1987/P507.PDF. [Pages 4 and 15]

[114] Shashi Shekhar, Hui Xiong, and Xun Zhou, editors. *Encyclopedia of GIS*. Springer, 2017. ISBN 978-3-319-17884-4. doi: 10.1007/978-3-319-17885-1. URL https://doi.org/10.1007/978-3-319-17885-1. [Page 14]

[115] Zhenghua Shu, Hong Li, Guodong Liu, Qing Xie, and Lvming Zeng. Application of gis in telecommunication information resources management system. In *Information Management, Innovation Management and Industrial Engineering (ICIII), 2011 International Conference on*, volume 1, pages 401–404. IEEE, 2011. [Page 2]

[116] Darius Sidlauskas, Simonas Saltenis, Christian W. Christiansen, Jan M. Johansen, and Donatas Saulys. Trees or grids?: indexing moving objects in main memory. In *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*, pages 236–245, 2009. doi: 10.1145/1653771.1653805. URL http://doi.acm.org/10.1145/1653771.1653805. [Page 14]

[117] Darius Sidlauskas, Christian S. Jensen, and Simonas Saltenis. A comparison of the use of virtual versus physical snapshots for supporting update-intensive workloads. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, pages 1–8, 2012. doi: 10.1145/2236584.2236585. URL http://doi.acm.org/10.1145/2236584.2236585. [Page 79]

[118] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 731–742, 2012. doi: 10.1145/2213836.2213946. URL http://doi.acm.org/10.1145/2213836.2213946. [Page 115]

[119] Chander Kumar Singh. *Geospatial Applications for Natural Resources Management*. CRC Press, 2018. [Page 2]

[120] Tomás Skopal, Michal Krátký, Jaroslav Pokorný, and Václav Snásel. A new range query algorithm for universal b-trees. *Inf. Syst.*, 31(6):489–511, 2006. doi: 10.1016/j.is.2004.12.001. URL https://doi.org/10.1016/j.is.2004.12.001. [Pages 100 and 129]

[121] statista. *Number of check-ins by registered members on Foursquare locations from August 2011 to August 2017 (in millions)*, 2018. https://www.statista.com/statistics/253838/number-of-check-ins-on-foursquare/. [Page 1]

[122] Emmanuel Stefanakis, Yannis Theodoridis, Timos K. Sellis, and Yuk-Cheung Lee. Point representation of spatial objects and query window extension: A new technique for spatial access methods. *International Journal of Geographical Information Science*, 11 (6):529–554, 1997. doi: 10.1080/136588197242185. URL https://doi.org/10.1080/136588197242185. [Pages 14, 75, 78, 90, 98, and 100]

[123] Alexandros Stougiannis, Mirjana Pavlovic, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. Data-driven neuroscience: enabling breakthroughs via innovative data management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 953–956, 2013. doi: 10.1145/2463676.2463677. URL http://doi.acm.org/10.1145/2463676.2463677. [Page 3]

[124] P. M. Suijker, I. Alkemade, M. P. Kodde, and A. E. Nonhebel. User requirements massive point clouds for esciences (wp1). *Delft University of Technology*, 2014. [Pages 5 and 18]

[125] Yufei Tao and Dimitris Papadias. Adaptive index structures. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 418–429, 2002. URL http://www.vldb.org/conf/2002/S12P02.pdf. [Page 15]

[126] Farhan Tauheed. *Scalable exploration of spatial data in large-scale scientific simulations*. PhD thesis, Ecole Polytechnique Federale de Lausanne, 2014. [Pages 2, 3, and 4]

[127] Farhan Tauheed, Laurynas Biveinis, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. Accelerating range queries for brain simulations. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 941–952, 2012. doi: 10.1109/ICDE.2012.56. URL https://doi.org/10.1109/ICDE.2012.56. [Pages 2, 4, 15, 24, 28, 29, 50, 78, 85, 86, and 100]

[128] Farhan Tauheed, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. OCTOPUS: efficient query execution on dynamic mesh datasets. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 1000–1011, 2014. doi: 10.1109/ICDE.2014.6816718. URL https://doi.org/10.1109/ICDE.2014.6816718. [Page 4]

[129] Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. Configuring spatial grids for efficient main memory joins. In *Data Science - 30th British International Conference on Databases, BICOD 2015, Edinburgh, UK, July 6-8, 2015, Proceedings*, pages 199–205, 2015. doi: 10.1007/978-3-319-20424-6_19. URL https://doi.org/10.1007/978-3-319-20424-6_19. [Pages 54 and 61]

[130] Yannis Theodoridis and Timos K. Sellis. Optimization issues in r-tree construction (extended abstract). In *IGIS '94: Geographic Information Systems, International Workshop on Advanced Information Systems, Monte Verita, Ascona, Switzerland, February 28 - March 4, 1994, Proceedings*, pages 270–273, 1994. doi: 10.1007/3-540-58795-0_54. URL https://doi.org/10.1007/3-540-58795-0_54. [Page 4]

[131] Yannis Theodoridis, Emmanuel Stefanakis, and Timos K. Sellis. Cost models for join queries in spatial databases. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 476–483, 1998. doi: 10.1109/ICDE.1998.655810. URL https://doi.org/10.1109/ICDE.1998.655810. [Page 16]

[132] Hermann Tropf and H. Herzog. Multimensional range search in dynamically balanced trees. *Angewandte Informatik*, 23(2):71–77, 1981. [Pages 11, 89, 100, 104, 117, and 129]

[133] Michael Ubell. The montage extensible datablade achitecture. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994.*, page 482, 1994. doi: 10.1145/191839.191939. URL http://doi.acm.org/10.1145/191839.191939. [Page 23]

[134] Peter van Oosterom, Oscar Martinez-Rubi, Milena Ivanova, Mike Hörhammer, Daniel Geringer, Siva Ravada, Theo Tijssen, Martin Kodde, and Romulo Goncalves. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers & Graphics*, 49:92–125, 2015. doi: 10.1016/j.cag.2015.01.007. URL https://doi.org/10.1016/j.cag.2015.01.007. [Pages 5, 18, 19, 100, 114, and 117]

[135] Sofia Berto Villas-Boas. Big data in firms and economic research. 2014. [Page 1]

[136] Fusheng Wang, Jun Kong, Lee Cooper, Tony Pan, Tahsin Kurc, Wenjin Chen, Ashish Sharma, Cristobal Niedermayr, Tae W Oh, Daniel Brat, et al. A data model and database for high-resolution pathology analytical image informatics. *Journal of pathology informatics*, 2, 2011. [Page 23]

[137] Aloysius Wehr and Uwe Lohr. Airborne laser scanning—an introduction and overview. *ISPRS Journal of photogrammetry and remote sensing*, 54(2-3):68–82, 1999. [Pages 113, 115, and 128]

[138] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009. doi: 10.14778/1687627.1687671. URL http://www.vldb.org/pvldb/2/vldb09-327.pdf. [Pages 115, 116, and 121]

[139] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *PVLDB*, 11(3):352–365, 2017. doi: 10.14778/3157794.3157803. URL http://www.vldb.org/pvldb/vol11/p352-zacharatou.pdf. [Page 3]

[140] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. Indexing for interactive exploration of big data series. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1555–1566, 2014. doi: 10.1145/2588555.2610498. URL http://doi.acm.org/10.1145/2588555.2610498. [Pages 16 and 86]

[141] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59, 2006. doi: 10.1109/ICDE.2006.150. URL https://doi.org/10.1109/ICDE.2006.150. [Page 3]

Mirjana Pavlović

Curriculum Vitae

EPFL IC IINFCOM DIAS BC 245
Station 14
1004 Lausanne
☎ *+41 21 69 36406*
✉ *mirjana.pavlovic@epfl.ch*
people.epfl.ch/mirjana.pavlovic

Education

2013–2019 **PhD in Computer and Communication Science**, EPFL, Switzerland.
- Thesis: Time- and Space-Efficient Spatial Data Analytics.
- Advisor: Prof. Anastasia Ailamaki.

2008–2009 **Master in Electrical and Computer Engineering**, University of Novi Sad, Serbia.
- GPA: 10/10.
- Major: Computing and Control Engineering/Applied Computer Science and Informatics.
- Thesis: Techniques for breaking dependencies between classes/modules.

2004–2009 **Bachelor in Electrical and Computer Engineering**, University of Novi Sad, Serbia.
- GPA: 9.66/10.
- Major: Computing and Control Engineering/Applied Computer Science and Informatics.
- Thesis: Realization of e-shopping through Internet using web services.

Work Experience

2013– **Research Assistant**, EPFL, Switzerland.
- Member of the Data-Intensive Applications and Systems laboratory.
- Research assistant in the context of the Human Brain Project for High Performance Analytics and Computing Platform.
- Working on time- and space-efficient solutions for spatial data management analytics.

2016 **Research Intern**, SAP SE, Germany.
- Part of the HANA Spatial group.
- Designed and implemented a prototype for point cloud data management.

2012–2013 **Scientific Assistant**, EPFL, Switzerland.
- Member of the Data-Intensive Applications and Systems laboratory.
- Designed and implemented an approach for the spatial join of massive datasets with contrasting density.

2010–2012 **Software Engineer**, Schneider Electric DMS NS, Serbia.
- Member of the Cross-platform application services team.
- Worked on the back-end systems for smart grid applications.

2009 **Intern**, Schneider Electric DMS NS, Serbia.
- Designed a set of techniques that enable efficient unit test execution in the legacy code systems.
- Evaluated the proposed solution in one of Schneider Electric DMS NS subsystems.

2008 **Intern**, Lanaco Information Technologies, Bosnia and Herzegovina.
- Designed and implemented information systems for library and financial management.

Publications

2018 M. Pavlovic, K. Bastian, H. Gildhoff and A. Ailamaki. Dictionary Compression in Point Cloud Data Management. To appear in ACM TSAS Special Issue.

2018 M. Pavlovic, D. Sidlauskas, T. Heinis and A. Ailamaki. QUASII: QUery-Aware Spatial Incremental Index. EDBT 2018.

2017 M. Pavlovic, K. Bastian, H. Gildhoff and A. Ailamaki. Dictionary Compression in Point Cloud Data Management. SIGSPATIAL 2017.

2016 M. Pavlovic, E. Tzirita Zacharatou, D. Sidlauskas, T. Heinis and A. Ailamaki. Space Odyssey - Efficient Exploration of Scientific Data. ExploreDB@SIGMOD 2016.

2016 M. Pavlovic, T. Heinis, P. Karras, F. Tauheed and A. Ailamaki. TRANSFORMERS: Robust Spatial Joins on Non-Uniform Data Distributions. ICDE 2016.

2016 M. Pavlovic, F. Tauheed, T. Heinis and A. Ailamaki. GIPSY: Joining Spatial Datasets with Contrasting Density. SSDMB 2013.

2013 A. Stougiannis, F. Thauheed, M. Pavlovic, T. Heinis, and A. Ailamaki. Data-driven Neuroscience: Enabling Breakthroughs Via Innovative Data Management. SIGMOD 2013.

2013 T. Heinis, F. Tauheed, M. Pavlovic and A. Ailamaki. Enabling Scientific Discovery Via Innovative Spatial Data Management. Data Engineering Bulletin 2013.

Awards and Honors

2018 EPFL Teaching assistant award.

2017 Best Paper Award for the paper "Dictionary Compression in Point Cloud Data Management", ACM SIGSPATIAL 2017.

2015 Selected to represent 1 out of 12 HBP sub-projects during the HBP Showcase.

2008–2009 Fellow of Schneider Electric DMS NS company.

2006–2009 Fellow of Power Utility of the Republic of Srpska company.

2005–2009 Republic of Srpska Government Fellowship.

2004–2009 University and faculty award for excellent results, University of Novi Sad.

2004 Student of Generation, High school Gymnasium Filip Višnjić, Republic of Srpska.

Conference Presentations and Invited Talks

2018 "Efficient Point Cloud Processing", IC research day, EPFL.

2018 "QUASII: QUery-Aware Spatial Incremental Index, EDBT.
2017 "Dictionary Compression in Point Cloud Data Management", ACM SIGSPATIAL.
2016 "Dictionary Compression in Point Cloud Data Management", SAP SE.
2016 "Space Odyssey - Efficient Exploration of Scientific Data", ExploreDB@SIGMOD.
2016 "Robust Spatial Joins on Non-Uniform Data Distributions", ICDE.
2015 "Data (and) Science: Adaptive Data Exploration", HBP Summit.
2013 "GIPSY: Joining Spatial Datasets with Contrasting Density", SSDMB.

Service

2018 Reviewer for The ACM Transactions on Spatial Algorithms and Systems journal.

In The News

2018 *ACM SIGSPATIAL Best Paper Award for DIAS Researchers*, Ecocloud News.

Teaching Experience

2015–2018 Introduction to database systems
2017 Programming
2014–2015 Information, Computation, Communication
2014 Programming I
2013 Programming

Languages

English Full professional proficiency
German Beginner
Serbian Native

References

Available upon request.

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**